# ¿Que es JS Reactor Tags  v0.0.3 ?

## Prologo:

**JS Reactor Tags** (jsReTag) , es un modulo desarrollado en Javascirpt que permite la interpretación reactiva y recursiva de etiquetas Hypertext Markup Language (HTML) con parametrizacion y cuerpo personalizado. Permitiendo así crear componentes reutilizables con diferentes finalidades de manera mas amigable y sencilla en comparación a como se haría si se utilizase JavaScrit en su esencia pura.

Js Reactor Tags es una pequeña pero poderosa herramienta que se integra al ecosistema ya existente de JavaScript y HTML  permitiendo una coexistencia consistente.

Las tareas encargadas a Js ReTag , son aquellas que requieran de la creación de componentes de cuerpo dinámico , creación por recursividad de componentes, de utilización de contextos de datos por identificación de origen unica, de la capacidad de adquisición de generar herencias de datos,de bloques de iteración por ciclos controlados , construcción por llamadas a apis y otras mas.

---------***INDICE DE DOCUMENTACION.***----------

*¿Que es JS Reactor Tags  v0.0.3 ?*

*Prologo*


*epilogo*

---

# 1. *Creación y utilización.*

La creación y utilización de componentes js ReTags es muy sencilla ya que el catalizador principal de Js ReTags se encargara de todo.

## 1.1 Enlazando Js Reactor Tags.

Para enlazar Js Reactor Tags con nuestro documento , debemos copiar el archivo “app.js” en nuestro directorio, este archivo contendrá el corazón catalizador que permitirá la reactividad de las etiquetas personalizadas, este enlace se realizara mediante la etiqueta clásica de <script src=””> en el documento HTML.
Listo, nuestro archivo es ahora un archivo compatible con Js ReactorTags!.

## 1.2 Enlazando componentes.

Para la creación de un componente js ReTag lo primero que debemos hacer es crear un archivo de componente .js como por ejemplo “section_offer_card.js” y enlazar  a nuestro archivo HTML mediante una etiqueta clásica de <script src=””>, con ello nuestro componente ya estaría enlazado , listo para ser creado y utilizado.

## 1.3 Creación de cuerpo de componentes.

Para dar origen a nuestro componente y nuestra nueva etiqueta personalizada deberemos crear dentro del archivo del componente una nueva instancia de `ReacetiveComponent();` mediante el uso del declarador de ámbito global “var” o en su defecto omitiendo declarador obteniendo el mismo resultado.

Con uso de declarador var

```
var section_offer_card = new ReacetiveComponent();
```

Sin uso de declarador var

```
section_offer_card = new ReacetiveComponent();
```

lo siguiente que debemos hacer es crear el cuerpo de nuestro componente mediante la asignación de una función con retorno de una cadena de string interpreta mediante Backstitch (``) al método `exportShape()`.

```
var section_offer_card = new ReacetiveComponent();

section_offer_card.exportShape() = function (data) {
    return `
      <div class="article-ticket-offer-body-card" style="background-image: url('offer.jpg')">
        <div class="article-ticket-offer-offert">-50% OFF</div>
        <a class="article-ticket-offer-body-anchor" href="#">
          <div class="article-ticket-offer-body-country">Argentina</div>
          <div class="article-ticket-offer-body-city">Buenos Aires</div>
        </a>
      </div>
    `;
};
```

*Importante: los nombre de los componentes deben de respetar la convención de lowercase asi también toda referencia a ellos dentro del ecosistema.*

## 1.4 Uso de componentes.

Al crear nuestro componente hemos creado a su vez una nueva etiqueta reactiva para nuestro proyecto , las etiquetas adoptan automáticamente el nombre de nuestro componente , es decir para el caso anterior , disponemos de una nueva etiqueta HTML llamada <section_offer_card/>, no obstante como todo evento reactivo , necesitamos un núcleo , el núcleo de la etiqueta , sera la propiedad “reactive” de JS Reactor Tags

```
< section_offer_card reactive>
   </section_offer_card>
```

JS Reactor Tags es compatible con self-closing tag de HTML

```
< section_offer_card reactive />
```

*Importante: los nombre de las propiedades deben de repetar la convención de lowercase asi también toda referencia a ellas dentro del ecosistema.*

En caso de que nos olvidemos dotar de la capacidad de un cuerpo a nuestro componente , jsReTag nos brindara por medio de su corazón reactivo avisos de importancia en tiempo real, permitiendo localizar errores humanos al momento de generar nuestro componentes , para este caso jsReTag nos dará el siguiente mensaje:

```
⚠ Js Reactor Tags say [WARM] :
    [flights_section] : no se ha encontrado el metodo exportShape()
```

# 1.5 Reutilización de componentes.

Js Reactor Tags permite la reutilización de componentes, veamos un ejemplo practico agregando múltiples veces el mismo componente jsReTag:

mi_componente.js

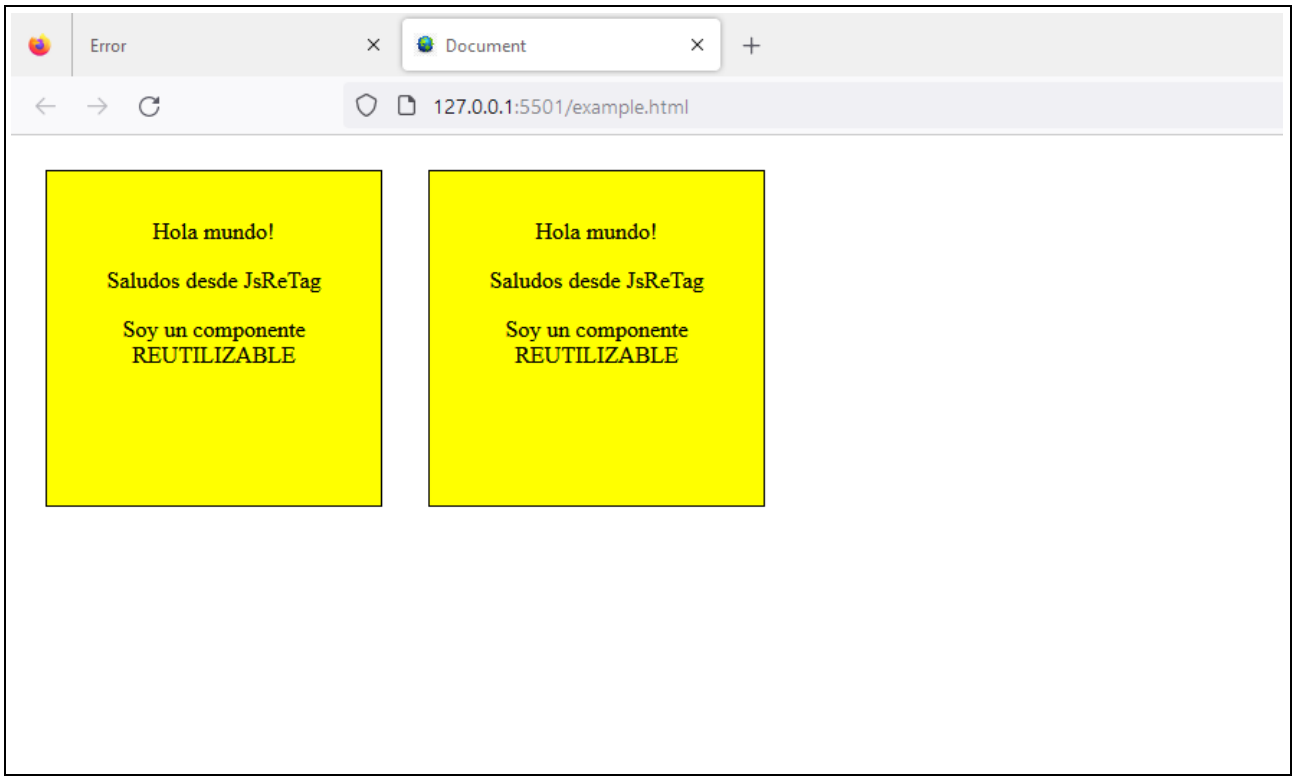
```
mi_componente = new ReacetiveComponent();

mi_componente.exportShape = function () {
    return `
        <div>
            <p>Hola mundo!</p>
            <p>Saludos desde JsReTag</p>
            <p>Soy un componente REUTILIZABLE</p>
        </div>
    `;
};
```

index.html

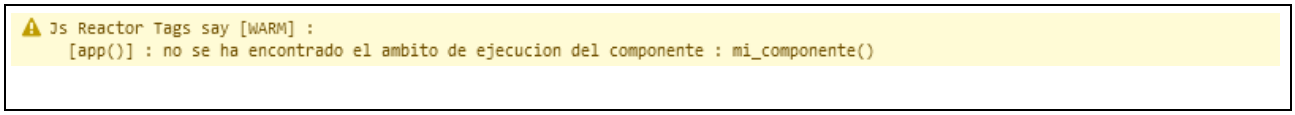
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
        <script src="./app/app.js"></script>
        <script src="./app/mi_componente.js"></script>
        <style>
            body {
                display: flex;
            }
            div {
                background-color: yellow;
                height: 200px;
                width: 200px;
                margin: 1rem;
                padding: 1rem;
                text-align: center;
                border: 1px solid black;
            }
        </style>
    </head>

    <body>
```

```
    <mi_componente reactive></mi_componente>
    <mi_componente reactive></mi_componente>
  </body>
</html>
```

resultado del user interface en el navegador web:




En caso de ingresar una etiqueta que no pertenece a un componente existente , jsReTag nos enviara un mensaje similar al siguiente:



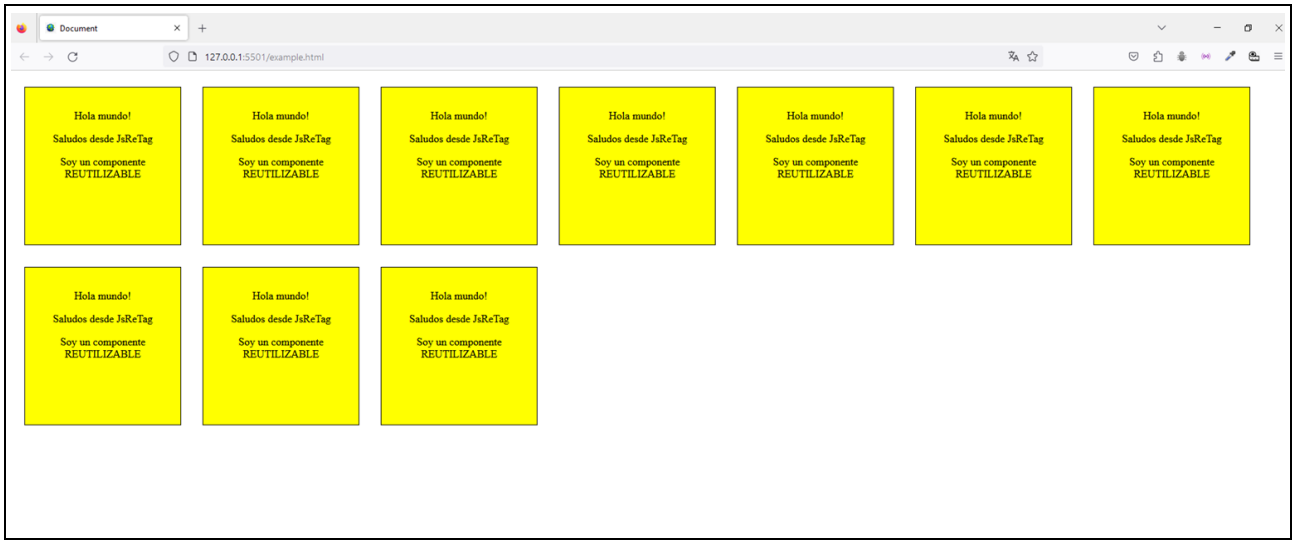

## 1.6 Reutilización de componentes por iteración de propiedad.

Js React Tags nos provee entre otras , de propiedades especificas para trabajar y hacer mas sencillo el manejo de nuestros componentes, en este caso hablaremos de la propiedad [iteration].
La propiedad [iteration] permite que un componente se duplique la cantidad de veces que lo requiramos y se lo indiquemos como valor de la propiedad.

Veamos un ejemplo sencillo reutilizando nuestro componente anterior “mi_componente” y agregándole la propiedad [iteration] asignada en 10 para crear este numero de elementos en nuestro documento :

```
<mi_componente reactive iteration="10"></mi_componente>
```

resultado del user interface en el navegador web:




# 2. *Tipos de componentes y características importantes.*

## Tipos de componentes.

En JsReTag existen inicialmente 2 tipos de componentes , los componentes del tipo Artesano o Shape y los componentes del tipo Constructores o Builders. Hasta ahora solo hemos conocido los componentes del tipo Shape, mas adelante veremos componentes del tipo Builder que permiten construir secciones completas por ejemplo mediante el uso de un link a una Api Rest entre otros.

## Características importantes de los componentes.

En JsReTag los componentes no solo son plantillas estáticas, si no que también podemos utilizar recursos de datos, contextos compartidos y apis rest para gestionarlas.

# 3. *Profundizando en los elementos Artesanos o Shape.*

Vamos a conocer algunas de las características de los elementos del tipo Shape

## 3.1 Creando un Resource en nuestro componentes

Los Resource son una característica de los componentes  tanto del tipo Shape como del tipo Builder en JsReTag, nos permiten tener separada la lógica de instancia de datos de la presentación de un cuerpo de componente JsReTag , esta lógica la podremos utilizar mas adelante , ahora veremos como construirla dentro de nuestro componente  mediante la asignación de una función con retorno de datos al metodo createResource() veamos un ejemplo practico:

mi_componente.js

```
mi_componente.createResource = function () {
```

```
    let amigos = [
        {
            nombre: "PABLITO",
            puedeCargar: "10",
        },
        {
            nombre: "JUANCITO",
            puedeCargar: "5",
        },
    ];
    let manzanas = Math.floor(Math.random() * 11);
    let datos = {};
    if (manzanas > 5) {
        datos = amigos[0];
    } else {
        datos = amigos[1];
    }
    datos.carga = manzanas;
    return datos;
};
```

## 3.2 Utilizando la característica Resource en nuestros componentes de tipo Shapes.

Podemos usar un Resource mediante el aviso de recepción del parámetro `(data)` en nuestra función de expresión que es asignada a `exportShape()` y la propiedad `use_data_resource` que deberá contener el valor de activación `"true"` en nuestra etiqueta jsReTag, veamos un ejemplo:

mi_componente.js

```
mi_componente.createResource = function () {
    let amigos = [
        {
            nombre: "PABLITO",
            puedeCargar: "10",
        },
        {
            nombre: "JUANCITO",
            puedeCargar: "5",
        },
    ];
    let manzanas = Math.floor(Math.random() * 11);
    let datos = {};
    if (manzanas > 5) {
        datos = amigos[0];
    } else {
        datos = amigos[1];
    }
    datos.carga = manzanas;
```

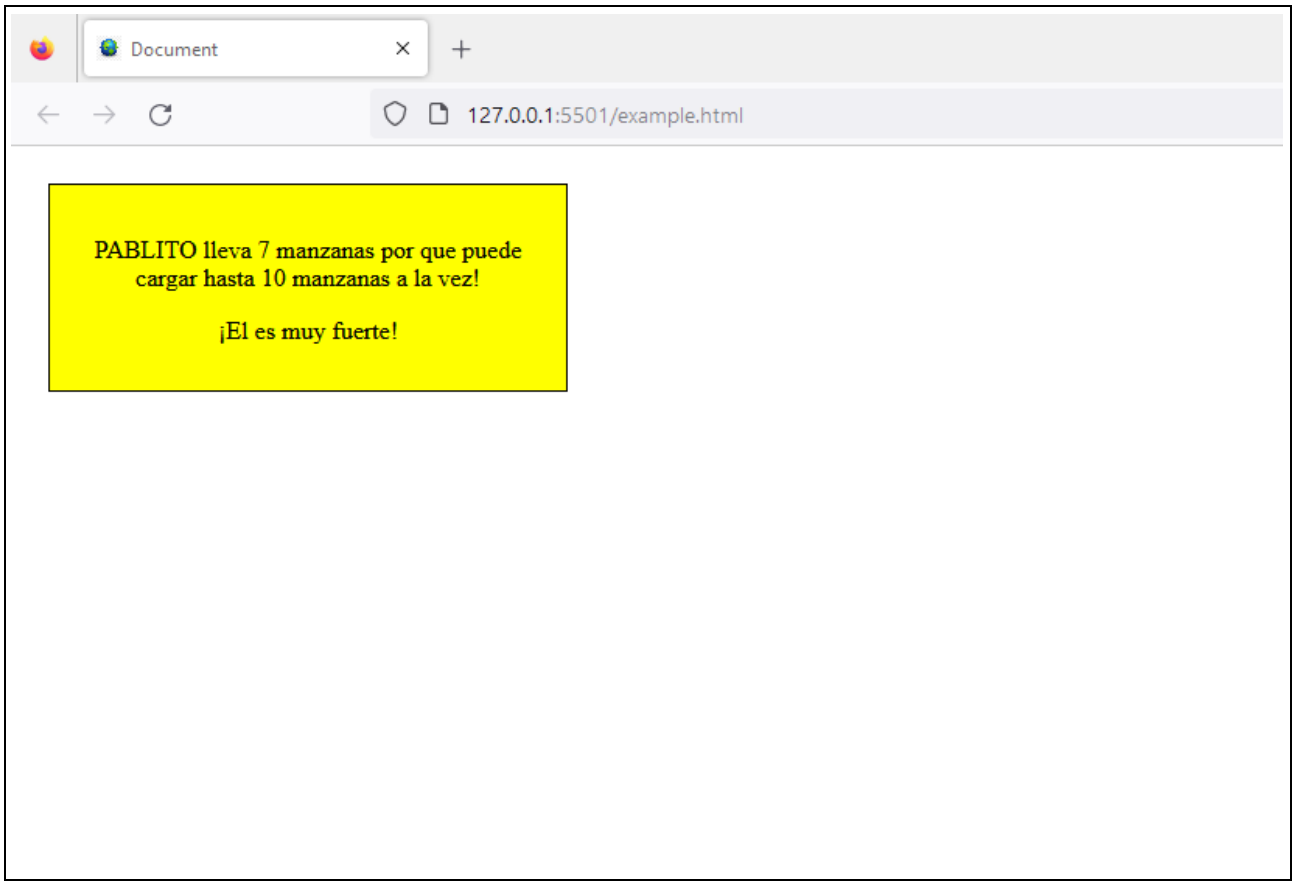
```
    return datos;
};

mi_componente.exportShape = function (data) {
    return `
        <div>
         <p>${data.resource.nombre} lleva ${data.resource.carga}  manzanas por que puede cargar hasta
${data.resource.puedeCargar} manzanas a la vez!</p>
        <p>¡El es muy fuerte!</p>

        </div>
    `;
};
```

index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
        <script src="./app/app.js"></script>
        <script src="./app/mi_componente.js"></script>
        <style>
            body {
                display: flex;
                flex-wrap: wrap;
            }
            div {
                background-color: yellow;
                height: 100px;
                width: 300px;
                margin: 1rem;
                padding: 1rem;
                text-align: center;
                border: 1px solid black;
            }
        </style>
    </head>

    <body>
        <mi_componente
            reactive
            iteration="1"
            use_data_resource="true"
            fixed_use_data="true"
        ></mi_componente>
```

```
    </body>
</html>
```

resultado del user interface en el navegador web:




Al utilizar la propiedad use_data_resource="true" dentro de la etiqueta jsRetag el parametro recibido sera un contenedor para los datos de resource solicitados, pudiéndose acceder a ellos mediante la siguiente nomenclatura data.resource.

Pero , ¿que sucede si indicamos la propiedad de iteración en 3 en nuestra etiqueta jsReTag ?

## 3.3 Shapes dinámicas con datos calculados fijados y no fijados.

Ahora que conocemos como manipular datos en nuestros componentes vamos a aprender a diferencias y manipular los Shapes dinámicos con datos calculados fijo y no fijos.
Siguiendo con el ejemplo anterior veremos que si agregamos la propiedad iteration="3" obtendremos un shape por ciclo con datos calculados dinámicos, es decir que cada caso sera calculado antes de ser enviado a nuestro cuerpo de componente.

```
    <mi_componente
```

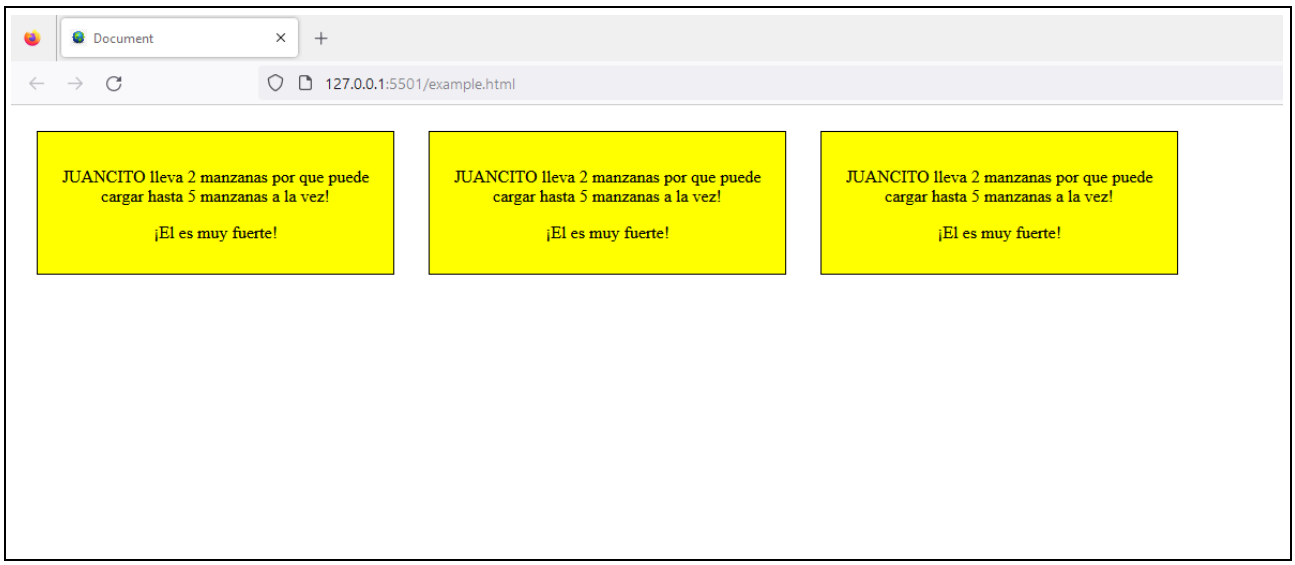
```
        reactive
        iteration="3"
        use_data_resource="true"
    ></mi_componente>
```

resultado del user interface en el navegador web:


Pero si queremos podemos emplear la propiedad y el concepto Shape por ciclo con datos calculados estáticos mediante el uso de la propiedad fixed_use_data la cual debe poseer el valor de activación “true”, Veamos un ejemplo:

```
<mi_componente
        reactive
        iteration="3"
        use_data_resource="true"
        fixed_use_data="true"
    ></mi_componente>
```

resultado del user interface en el navegador web:




Aquí es donde comenzamos a ver el potencial de Js Reactior Tags, aun queda mucho por descubrir.

## 3.4 Utilizando la característica Api Rest en nuestros componentes de tipo Shapes.

JsReTag nos presenta la característica de obtener datos desde una api para trabajar en nuestros componentes de tipo Shapes y Builders, para ello utilizaremos las propiedades `url_api` que deberá contener el valor de la url de la api a ser consultada y `use_data_api` que deberá contener el valor de activación `"true"` en nuestra etiqueta.

index.html

```
<mi_componente
        reactive
        url_api="https://jsonplaceholder.typicode.com/posts?id=10"
        use_data_api="true"
    ></mi_componente>
```

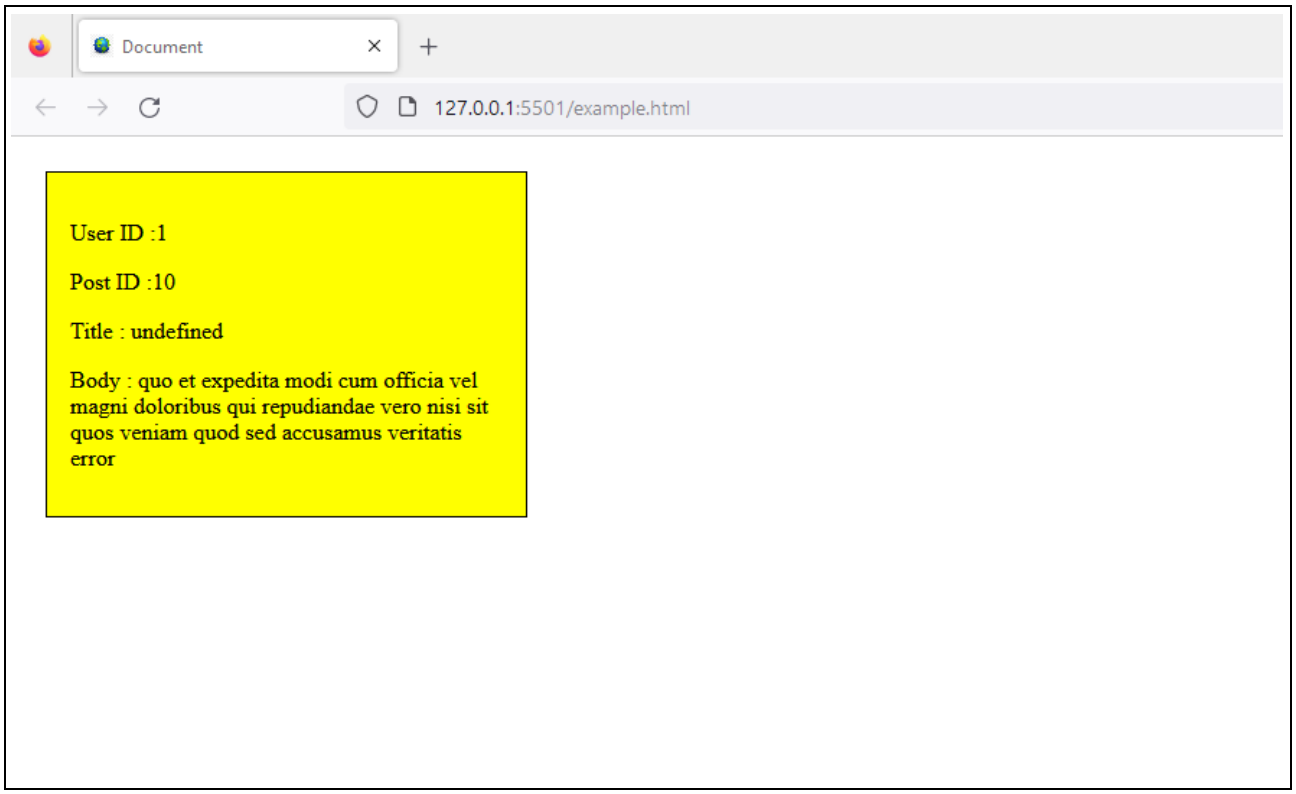
Veamos un ejemplo practico conectándonos a la ApiRest de jsonplaceholder.

mi_componente.js

```
mi_componente = new ReacetiveComponent();

mi_componente.exportShape = function (data) {
    console.log(data.api);
    return `
      <div>
       <p>User ID :${data.api[0].userId}</p>
       <p>Post ID :${data.api[0].id}</p>
       <p>Title : ${data.api[0].tittle}</p>
       <p>Body : ${data.api[0].body}</p>
      </div>
    `;
```

```
};
```

resultado del user interface en el navegador web:




## 3.5 Utilizando la característica de Contextos en nuestros componentes de tipo Shapes.

Los contexto nos permiten guardar y obtener datos desde diferentes orígenes para utilizarlos en la construcción de cuerpos de otros componentes Js Rector Tags mediante el uso de la propiedad use_data_context para consumir un contexto , esta propiedad deberá contener el valor del identificador de contexto , y el método createContext() para crear un contexto , donde los parámetros que recibirá son el nombre del contexto y los datos a guardar en el contexto , veamos un ejemplo practico:

mi_componente.js

```
mi_componente = new ReacetiveComponent();

mi_componente.createResource = function () {
    let amigos = [
        {
            nombre: "PABLITO",
            puedeCargar: "10",
        },
        {
            nombre: "JUANCITO",
            puedeCargar: "5",
```

```
        },
    ];
    let manzanas = Math.floor(Math.random() * 11);
    let datos = {};
    if (manzanas > 5) {
        datos = amigos[0];
    } else {
        datos = amigos[1];
    }
    datos.lleva = manzanas;
    mi_componente.createContext("cantidadCargada", { lleva: datos.lleva });
    mi_componente.createContext("cantidadSoportada", {
        lleva: datos.puedeCargar,
    });
    return datos;
};

mi_componente.exportShape = function (data) {
    return `
    <div>
    <h3>Soy el componente "mi_componente"</h3>
        <p>Quien las lleva?</p>
        <p> ${data.resource.nombre}</p>
    </div>
`;
};
```

informe1.js

```
informe1 = new ReacetiveComponent();

informe1.exportShape = function (data) {
    return `
       <div>
        <h3>Soy el componente "informe1"</h3>
        <p>Cuantas manzanas lleva? </p>
        <p> ${data.context.lleva} manzanas</p>
       </div>
 `;
};
```

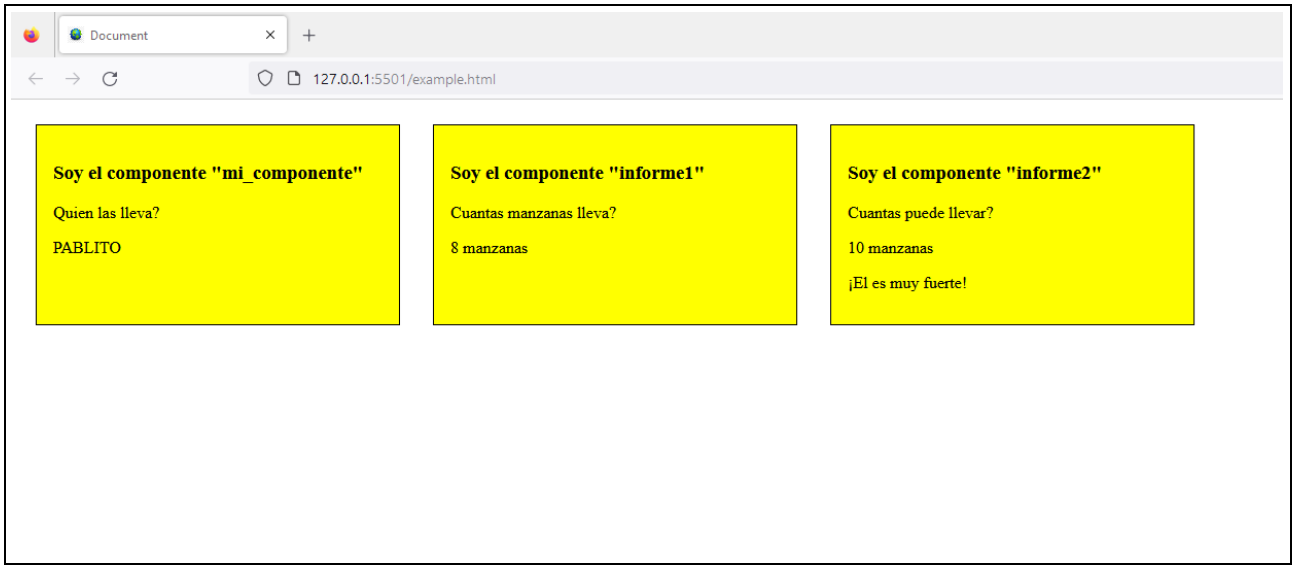
informe2.js

```
informe2 = new ReacetiveComponent();

informe2.exportShape = function (data) {
    return `
      <div>
       <h3>Soy el componente "informe2"</h3>
       <p>Cuantas puede llevar? </p>
       <p> ${data.context.lleva} manzanas</p>
       <p>¡El es muy fuerte!</p>
      </div>
 `;
};
```

index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
        <script src="./app/app.js"></script>
        <script src="./app/mi_componente.js"></script>
        <script src="./app/informe1.js"></script>
        <script src="./app/informe2.js"></script>
        <style>
            body {
                display: flex;
                flex-wrap: wrap;
            }
            div {
                background-color: yellow;
                width: 300px;
                margin: 1rem;
                padding: 1rem;
                border: 1px solid black;
            }
        </style>
    </head>

    <body>
        <mi_componente reactive use_data_resource="true"></mi_componente>
        <informe1 reactive use_data_context="cantidadCargada"></informe1>
        <informe2 reactive use_data_context="cantidadSoportada"></informe2>
    </body>
</html>
```

resultado del user interface en el navegador web:




Esto permite que los componentes de jsReTag puedan comunicarse entre si.

En caso de intentar solicitar acceso a un Contexto el cual no existe Js Reactor Tags nos enviara un mensaje similar al siguiente.

```
⚠ Js Reactor Tags say [WARM] :
    [app()] : El contexto datos_usuario_registro no existe.
```

# *4. Conociendo los componentes del tipo Constructores o Builders.*

Ya que hemos conocido las características principales de los componentes del tipo Shape , pasaremos a interiorizarnos en los componentes del tipo Builder, estos componentes están orientados para crear estructuras robustas a bajo coste de manipulación.

## 4.1 La potencia de los Builder sobre las Api Rest.

Para la creación de un builder cuya finalidad es la de construir una estructura robusta de datos mediante la obtención de los mismos de una fuente externa , necesitaremos primero seguir los mismos pasos para crear un componente del tipo Shape , pero en vez de utilizar el método `exportShape()` utilizaremos el método destinado a la construcción mediante builders , usaremos el método `exportBuilder()` que también recibirá un parámetro data con los datos necesarios para que este constructor pueda comenzar a trabajar , en tanto a la etiqueta reactiva deberá poseer la propiedad `build_by_api` que contendrá el valor de activación `"true"` y la propiedad `url_api` que

contendrá la ubicación de la Api Rest a ser consumida,para controlar la cantidad de construcciones , podemos emplear la propiedad iteration asignándole un máximo de construcciones según le indiquemos, veamos un ejemplo practico:

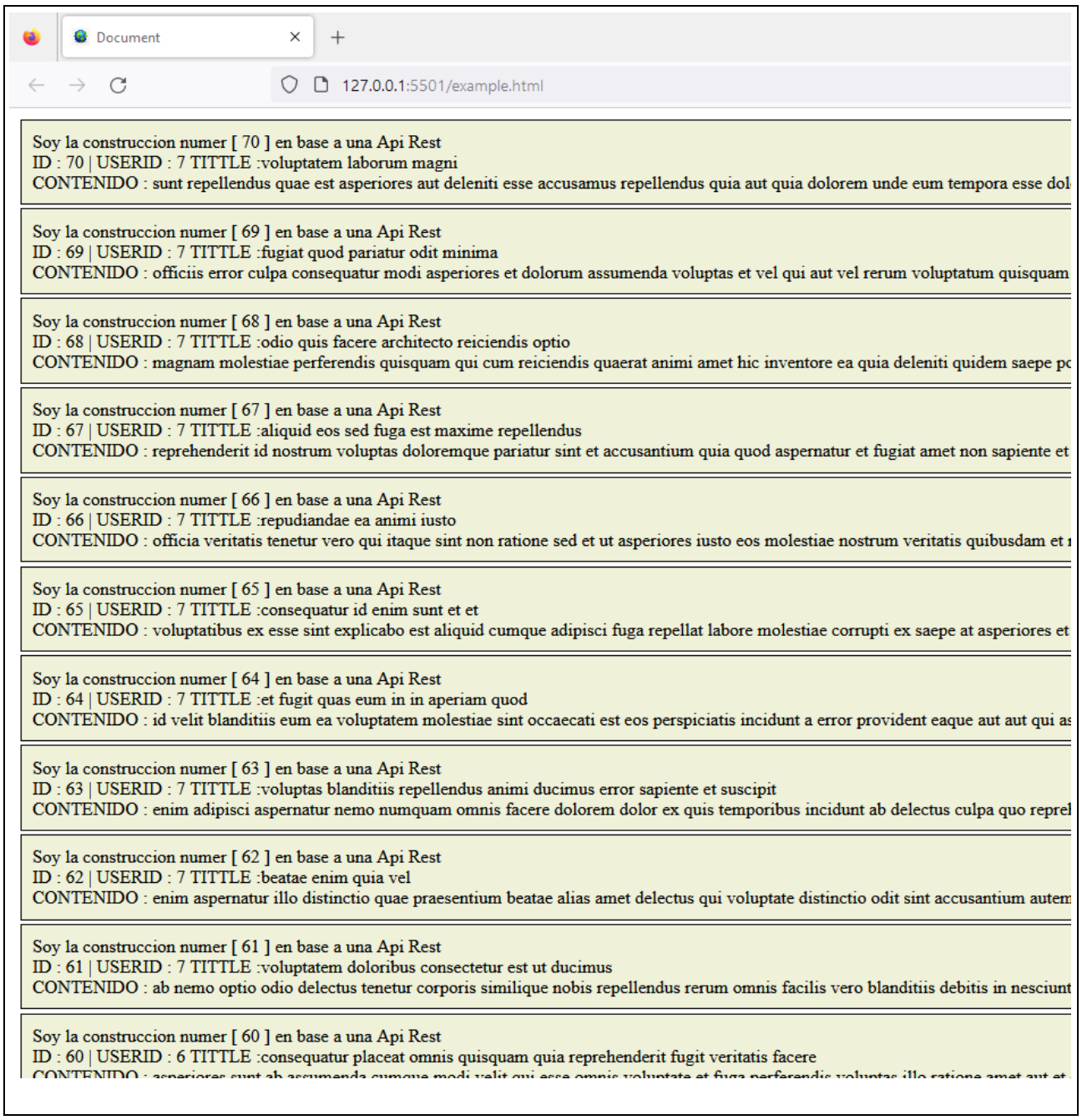
mi_componente.js

```
mi_componente = new ReacetiveComponent();

mi_componente.exportBuilder = function (data) {
    console.log(data);
    return `
    <div>
        <p>
        Soy la construccion numer [ ${data.id} ] en base a una Api Rest <br>
         ID : ${data.id} | USERID : ${data.userId} TITTLE :${data.title} <br>
         CONTENIDO : ${data.body}
       asdasd
         </p>
    </div>
`;
};
```

*index.html*

```
    <mi_componente
            reactive
            url_api="https://jsonplaceholder.typicode.com/posts"
            iteration="70"
            build_by_api="true"
        ></mi_componente>
```

resultado del user interface en el navegador web:




Hemos visto como con unas pocas lineas de código hemos generado una estructura de código robusta.

## 4.2 Los Builder build_by_context  y build_by_resource

En tanto a las demás propiedades de los builders bajo el mismo concepto de creación y uso , js Reactor Tags nos provee de las siguientes propiedades:build_by_context nos permitirá crear estructuras en base a datos guardados por cualquier componente mediante uso de createContext() , el valor de la propiedad deberá contener el nombre del contexto el cual sera usado por el builder.

build_by_resource nos permitirá crear estructuras en base a datos guardados en el contexto del componente , el valor de la propiedad deberá contener el valor de activación "true".

# 4.3 Herencias de api sobre builders para construccion recursiva

Antes que nada, debemos saber que Js Reactor Tags es compatible con la reactividad recursiva. Esto significa que cada etiqueta reactiva contenida dentro otra etiqueta reactiva también será reconocida por el catalizado y el corazón de js Reactor Tags independientemente de que sea del tipo Shape o Builder.

Sabiendo eso , la herencia de api sobre builders para construcción recursiva permite que un componente entregue como herencia automáticamente los datos recolectados por el llamado de una api para que los componentes reactivos en su interior puedan utilizarla.

Para ello el componente que entregue la herencia deberá contener dos propiedades dentro de su etiqueta jsReTag siendo una de ellas la propiedad url_api que contendrá el valor con la dirección de la api a ser consultada y la otra propiedad deberá ser use_api_data con la asignación del valor de activación "true", por otro lado , el o los componentes reactivos contenidos dentro que quieran hacer uso de los recursos de la api deberán poseer la propiedad build_by_api_herency con el valor de activación "true" veamos un ejemplo practico:

mi_componente.js

```
mi_componente = new ReacetiveComponent();

mi_componente.exportBuilder = function (data) {
    return `
    <div class="container">
        <div id="left" class="column-class">
        <h3>Esta columna contiene el componente informe1</h3>
        <h3>Mostrara el ID , USER ID y TUTILO del POST de cada elemento de la Api HEREDARA"</h3>
        <informe1 reactive iteration="max" build_by_api_herency="true" ></informe1>
        </div>
        <div id="right" class="column-class">
        <h3>Esta columna contiene el componente informe2</h3>
        <h3>Mostrara los datos del cuerpo del POST de cada elemento de la Api HEREDARA</h3>
        <informe2 reactive iteration="max" build_by_api_herency="true" ></informe2>
        </div>
    </div>
`;
};
```

informe1.js

```
informe1 = new ReacetiveComponent();

informe1.exportBuilder = function (data) {
    return `
    <div>
```

```
        <p>
            Soy la construccion numero [ ${data.id} ] en base a una Api Rest transferida por HERENCIA<br>
            ID : ${data.id} | USERID : ${data.userId} TITTLE :${data.title}
        </p>
    </div>
`;
};
```

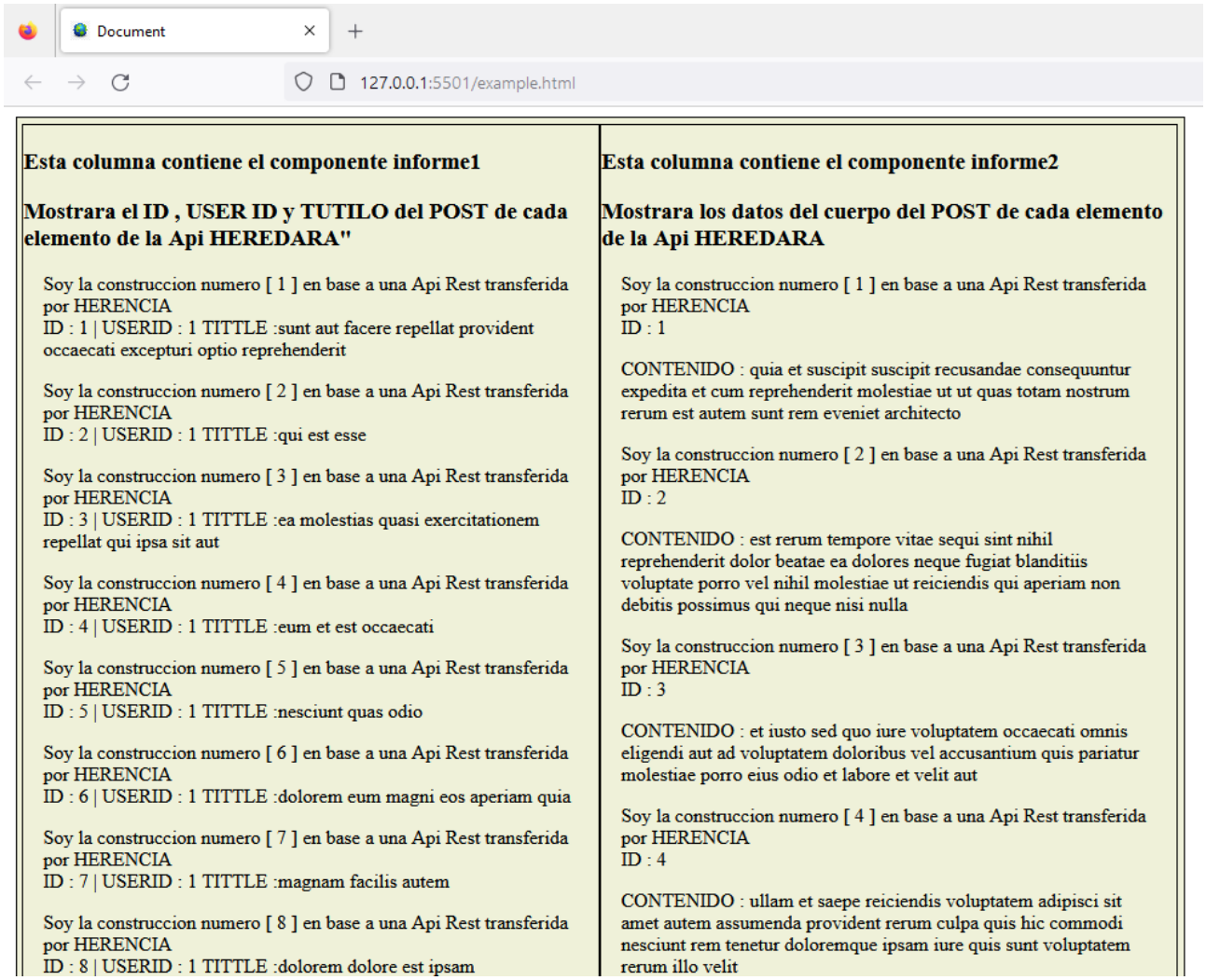
informe2.js

```
informe2 = new ReacetiveComponent();

informe2.exportBuilder = function (data) {
    return `
    <div>
        <p>
          Soy la construccion numero [ ${data.id} ] en base a una Api Rest transferida por HERENCIA<br>
           ID : ${data.id}
        </p>
        <p>CONTENIDO : ${data.body}</p>
    </div>
`;
};
```

Index.html

```
        <mi_componente
            reactive
            url_api="https://jsonplaceholder.typicode.com/posts"
            iteration="1"
            build_by_api="true"
        ></mi_componente>
```

resultado del user interface en el navegador web:




# 4. *Ciclo de vida de un componente y sus estados.*

## 4.1 El ciclo de vida de un componente.

Los componentes siguen un ciclo de vida similar al de las abstracciones del mundo real. Esto significa que un componente, por ejemplo, nace, se desarrolla y eventualmente deja de existir en algún punto de su ciclo de vida. A estas pequeñas instancias las llamamos 'estados'.

## 4.2 Estado de un componente.

Los estados son una parte fundamental que nos permite interactuar con los componentes y realizar acciones en el momento en que la existencia del estado está habilitada dentro de un contexto dado.

## 4.3 Composición de un ciclo de vida por estados.

Js Reactor Tags nos presenta la siguiente lista de estados ordenada y representando la forma en la transcurre un ciclo de vida de un componente:

```
componentWillCreate()
|- componentWillMount()
|- componentDidMount()
|- componentWillCreateAllReactiveChilds()
|  |- componentWillCreateReactiveChild()
|  |- componentDidCreateReactiveChild()
|- componentDidCreateAllReactiveChilds()
componentDidCreate()
```

veamos que momento del ciclo de vida intercepta cada estado que nos brinda Js Reactor Tags:

componentWillCreate() : intercepta el momento anterior de la creación completa de un componente esto incluye la construcción de las etiquetas reactivas recursivas contenidas en el.

ComponentWillMount() : intercepta el momento anterior en el que el cuerpo del componente es montado sobre el nodo DOM superior inmediato.

ComponentDidMount() : intercepta el momento posterior en el que el cuerpo del componente es montado sobre el nodo DOM superior inmediato.

ComponentWillCreateAllReactiveChilds() : intercepta el momento anterior en el que el componente comenzara la construcción recursiva de sus etiquetas reactivas contenidas en el

componentWillCreateReactiveChild() : intercepta el momento anterior a la creación de un componente Reactivo inmediato que este contenido dentro del cuerpo del componente contenido.

ComponentDidCreateReactiveChild() : intercepta el momento posterior a la creación de un componente Reactivo inmediato que este contenido dentro del cuerpo del componente contenido.

ComponentDidCreateAllReactiveChilds() : intercepta el momento posterior en el que el componente comenzara la construcción recursiva de sus etiquetas reactivas contenidas en El.

ComponentDidCreate() : intercepta el momento posterior de la creación completa de un componente esto incluye la construcción de las etiquetas reactivas recursividad contenidas en El.

## 4.4 Utilizando los estados de un ciclo de vida.

Para utilizar cualquiera de los estados de un ciclo de vida de un componente js Reactor Tags lo que debemos hacer es crear una función que contenga dentro de ella lo que queramos realizar al momento de concepción del estado asignándola al método relativo correspondiente.
veamos una ejemplo practico y sencillo que abarque el uso de todos los estados en un mismo componente:

mi_componente_recursivo.js

```
var mi_componente_recursivo = new ReacetiveComponent();

mi_componente_recursivo.exportShape = function (data) {
    return `
```

```
        Cuerpo de "mi_componente_recursivo"
    `;
};
```

index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
        <script src="./app/app.js"></script>
        <script src="./app/mi_componente.js"></script>
        <script src="./app/mi_componente_recursivo.js"></script>
    </head>
    <body>
        <mi_componente reactive iteration="1"></mi_componente>
    </body>
</html>
```

mi_componente.js

```
var mi_componente = new ReacetiveComponent();

mi_componente.componentWillCreate = function () {
    alert("El componente ha sido creado");
};

mi_componente.componentWillMount = function () {
    alert("El componente está a punto de montarse en el DOM");
};

mi_componente.componentDidMount = function () {
    alert("El componente se ha montado en el DOM");
};

mi_componente.componentWillCreateAllReactiveChilds = function () {
    alert("El componente está a punto de crear sus hijos reactivos");
};

mi_componente.componentWillCreateReactiveChild = function () {
    alert("El componente está a punto de crear un hijo reactivo");
};

mi_componente.componentDidCreateReactiveChild = function () {
    alert("El componente ha creado un hijo reactivo");
};

mi_componente.componentDidCreateAllReactiveChilds = function () {
```

```
    alert("El componente ha creado todos sus hijos reactivos");
};

mi_componente.componentDidCreate = function () {
    alert("El componente ha sido completamente creado");
};

mi_componente.exportShape = function (data) {
    return `
        <p>|----Cuerpo de "mi_componente"<p>
        <p>      
        |----<mi_componente_recursivo iteration="1" reactive> </mi_componente_recursivo></p>
    `;
};
```

# 4.5 Accediendo a la abstracción del componente desde sus estados.

Js React Tag nos permite acceder y manipular al objeto que contiene la abstracción del componente mismo , para ello simplemente deberemos agregar al método correspondiente del estado en cuestión la posibilidad de recibir como parámetro un objeto el cual sera la abstracción real del componente.

Veamos un ejemplo practico y sencillo:

example.html

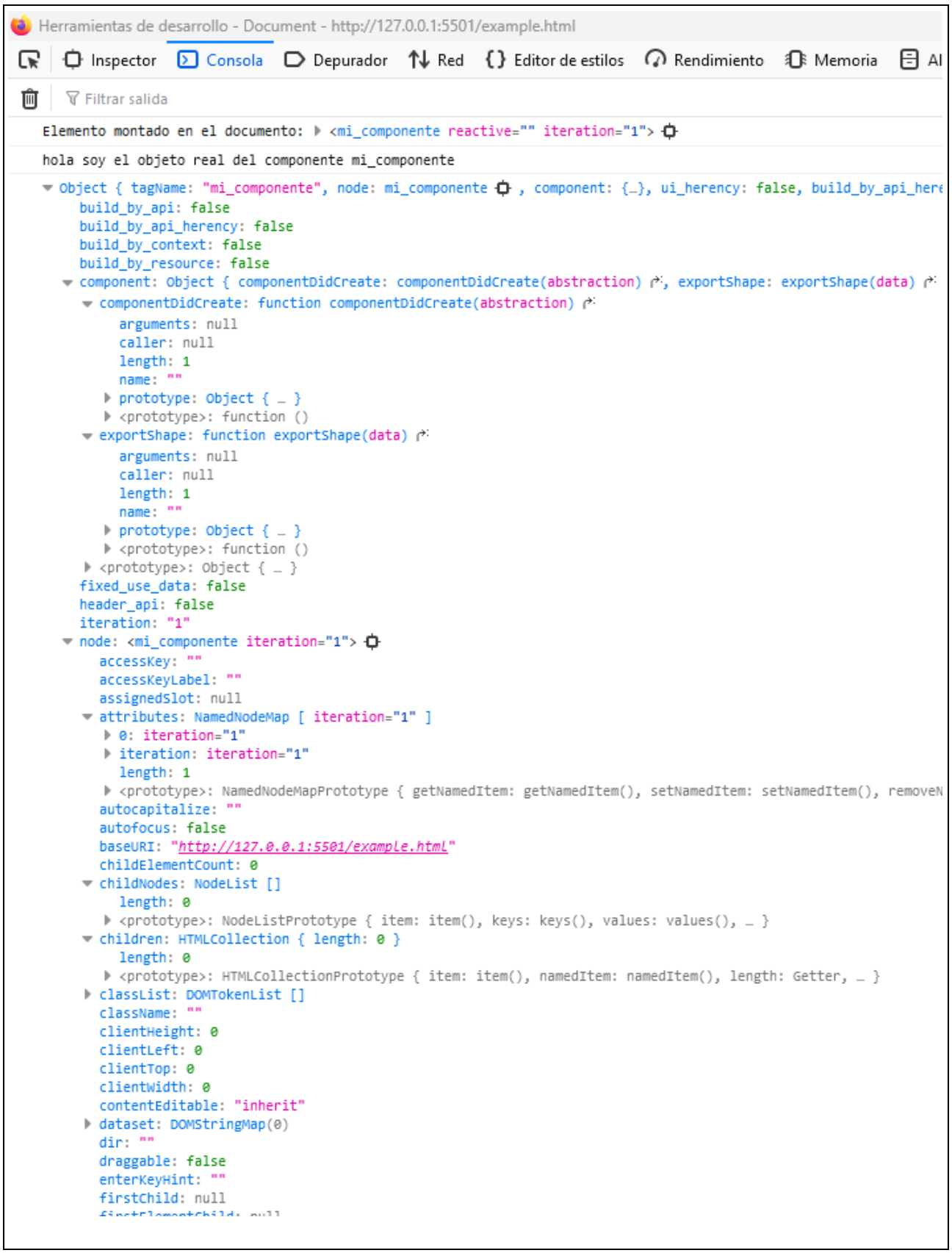
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
        <script src="./app/app.js"></script>
        <script src="./app/mi_componente.js"></script>
    </head>

    <body>
        <mi_componente reactive iteration="1"></mi_componente>
    </body>
</html>
```

mi_componente.html

```
mi_componente = new ReacetiveComponent();
mi_componente.componentDidCreate = function (abstraction) {
    console.log(
        `hola soy el objeto real del componente ${abstraction.tagName}`
    );
    console.log(abstraction);
```

```
};
mi_componente.exportShape = function (data) {
    return `

    `;
};
```

resultado de la consola del user interface en el navegador web:


Herramientas de desarrollo - Document - http://127.0.0.1:5501/example.html
Inspector
Consola
Depurador
Red
Editor de estilos
Rendimiento
Memoria
Filtrar salida
Elemento montado en el documento: <mi_componente reactive="" iteration="1">
hola soy el objeto real del componente mi_componente
Object { tagName: "mi_componente", node: mi_componente , component: {…}, ui_herency: false, build_by_api_here
build_by_api: false
build_by_api_herency: false
build_by_context: false
build_by_resource: false
component: Object { componentDidCreate: componentDidCreate(abstraction), exportShape: exportShape(data)
componentDidCreate: function componentDidCreate(abstraction)
arguments: null
caller: null
length: 1
name: ""
prototype: Object { … }
<prototype>: function ()
exportShape: function exportShape(data)
arguments: null
caller: null
length: 1
name: ""
prototype: Object { … }
<prototype>: function ()
<prototype>: Object { … }
fixed_use_data: false
header_api: false
iteration: "1"
node: <mi_componente iteration="1">
accessKey: ""
accessKeyLabel: ""
assignedSlot: null
attributes: NamedNodeMap [ iteration="1" ]
0: iteration="1"
iteration: iteration="1"
length: 1
<prototype>: NamedNodeMapPrototype { getNamedItem: getNamedItem(), setNamedItem: setNamedItem(), removeN
autocapitalize: ""
autofocus: false
baseURI: "http://127.0.0.1:5501/example.html"
childElementCount: 0
childNodes: NodeList []
length: 0
<prototype>: NodeListPrototype { item: item(), keys: keys(), values: values(), … }
children: HTMLCollection { length: 0 }
length: 0
<prototype>: HTMLCollectionPrototype { item: item(), namedItem: namedItem(), length: Getter, … }
classList: DOMTokenList []
className: ""
clientHeight: 0
clientLeft: 0
clientTop: 0
clientWidth: 0
contentEditable: "inherit"
dataset: DOMStringMap(0)
dir: ""
draggable: false
enterKeyHint: ""
firstChild: null

# Epilogo

consideraciones:
-Js Reactor tags permite objetos como parte del sistema de parámetros.
-Tener etiquetas JsReTag junto con etiquetas HTML es compatible.
-Los métodos de cuerpo de los componentes Shapes y Builder pueden contener lógica en su interior previo al retorno de los Backstitch.
-Muchos conceptos mas que son posible aplicar pero no están detallados el la presente versión de documentación.

motivación:

Este modulo se desarrollo a lo largo de unos pocos día principalmente para la adición a la entrega del proyecto solicitado por la capacitación formativa de CODO A CODO 4.0 y para ser utilizado como herramienta de ejemplo para las personas que recién estamos empezando en este hermoso mundo de la programación.

Agradecimientos:

A mi tutor a cargo Federico Liquin https://github.com/broko-de quien me acompaño a lo largo del trayecto formativo en la presente instancia.

*Creación y desarrollo:*

**Marucci Mauro Nahuel**

https://github.com/maruccimauro
marucci.mauro@gmail.com
https://www.linkedin.com/in/mauro-marucci/

**v 0.0.3**