# React 

*“A Javascript library for building user interfaces”*

# Introducción

- Documentación: https://reactjs.org/
- Desarrollada por **Facebook** y liberada en 2013.
- Librería para crear interfaces de usuario en el desarrollo de aplicaciones **SPA** (Single Page Applicattions).
- No es un **framework**, pero se ha creado un amplio ecosistema de librerías alrededor (estado, routing, ajax, …).
    - `npm install --save react`

# Principios básicos

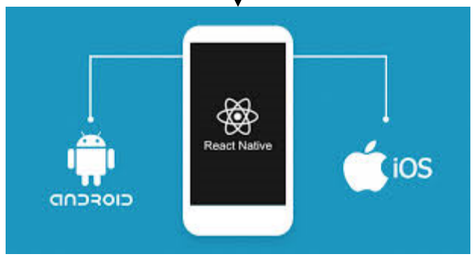
- **Declarativo:** diseñar la vista para cada estado, React actualiza de modo eficiente cuando cambian los datos.
- **Basada en componentes:** construir componentes encapsulados y componer interfaces más complejas.
- **Aprender una vez, escribir todo tipo de aplicaciones**
  - React-DOM: aplicaciones web
  - React-Native: aplicaciones móviles
  - React-360: aplicaciones de realidad virtual

# Principios básicos (II)

React

React-DOM

React-Native




React-360

# Elementos: React.createElement

- **Elemento:** pieza más pequeña en aplicaciones React que describe lo que vemos en pantalla.
- **React.createElement:** creando elementos React

```
const element = React.createElement('div', {className: 'container'}, 'Hello, World!');
```

- **ReacDOM.render:** Renderizando elementos en el DOM

```
<htm>
  <body>
    <div id='root'></div>
  </body>
</htm>;

ReactDOM.render(element, document.getElementById('root'));
```

# Elementos: JSX

- **JSX:** extensión de **Javascript** con sintaxis tipo XML(HTML).

```
const element = <div className="container">Hello, World!</div>;
```

- **Sugar sintax: Babel** transpila JSX en createElement.
- Usando expresiones Javascript en JSX entre llaves ({}).

```
const name = 'John Doe';
const className = 'container';
const element = <div className={className}>Hello, {name}!</div>;
```

- Atributos **camelCase**, **className** (class) y **htmlFor** (for).
- Elementos con **children**  y sin **children** <img/>

```
<div className="container">Hello, World!</div>
<img src="picture.jpg"/>
```

# Elementos: JSX (II)

- Renderizado condicional con el operador **&&**

```
{unreadMessages.length > 0 && <h2>You have {unreadMessages.length} messages.</h2>}
```

- Renderizado condicional con el operador ternario **?**

```
<div>The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.</div>
```

- Evitando el renderizado de un componente

```
if (!props.warn) return null;
```

# Elementos: JSX (III)

- Propagando atributos con el spread operator ...

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

- Props por defecto a true

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

# Componentes: props

- **Código (función o clase)** que recibe objeto de propiedades (**props**) por parámetro y devuelve elementos React.

```
const Welcome = (props) => <div>Hello, {props.name}</div>;
```

```
class Welcome extends React.Component {
  render() {
    return(<div>Hello, {this.props.name}</div>);
  }
}
```

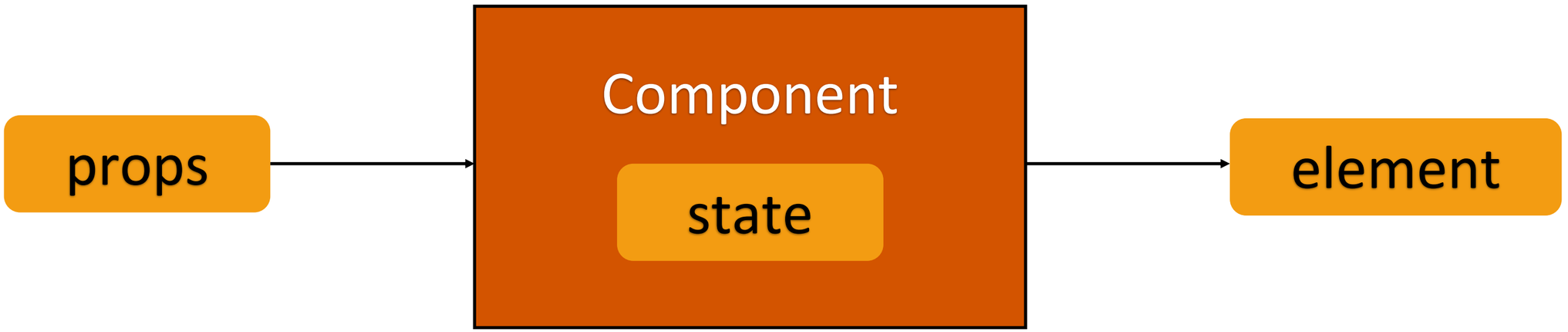
- Renderizando un componente (comienza por mayúscula).

```
const element = <Welcome name="John“/>;
```

- Un componente no debe modificar las props (función pura)!!!

# Componentes: estado y ciclo de vida

- **Estado:** Conjunto de datos internos que junto a las props definen el elemento renderizado en un momento dado.




- **No modificar estado directamente!!!**
- Actualizaciones de estado **asíncronas** (no fiarse del estado anterior).
- Las actualizaciones de estado se fusionan.

# Componentes: estado y ciclo de vida (II)

- **Definiendo** el estado de un componente (*this.state*)

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>It is {this.state.date.toLocaleTimeString()}</div>
    );
  }
}
```

# Componentes: estado y ciclo de vida (III)

- **Modificando** el estado de un componente (*this.setState*)

```
class Clock extends React.Component {
  constructor (props) {
    super(props);
    this.state = { date: new Date() };
  }

  tick = () => this.setState({date: new Date()});

  componentDidMount() {
    this.timerID = setInterval(this.tick, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  render () {
    return <div>It is {this.state.date.toLocaleTimeString()}</div>;
  }
}
```

# Componentes: estado y ciclo de vida (IV)

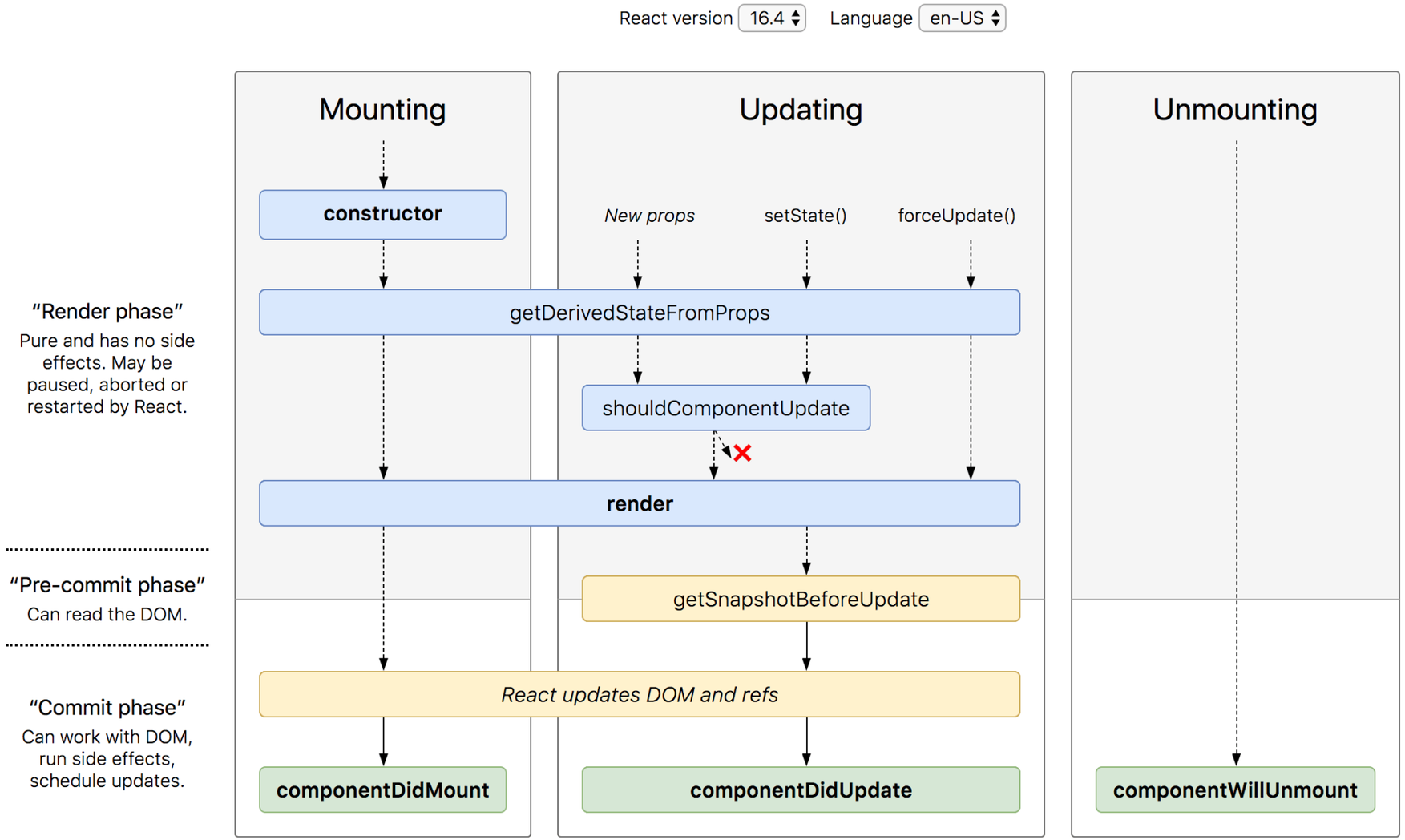
- **this.setState** provoca un renderizado del componente.

```
// Incorrecto
this.state.comment = 'Hello';

// Correcto
this.setState({ comment: 'Hello' });
```

- La modificación se fusiona con el estado actual.
- Si necesitamos calcular el nuevo estado en función del estado anterior podemos pasar una función.

```
// Incorrecto
this.setState({ counter: this.state.counter + this.props.increment });

// Correcto
this.setState((state, props) => ({ counter: state.counter + props.increment }));
```

# Componentes: estado y ciclo de vida (V)




# Componentes: eventos

- Se manejan de modo similar a los eventos de los elementos DOM.
  - Nombres de eventos en camelCase
  - Debemos pasar una función
  - **event.preventDefault**: evitamos comportamiento por defecto.
  - **SyntheticEvent:** wrapper sobre evento nativo del DOM.

```
// HTML
<button onclick="console.log()">Click here!</button>

// React
<button onClick={(event) => console.log(event)}>Click here!</button>
```

- No necesitamos llamar **addEventListener**.

# Componentes: eventos (II)

- **this** en métodos de clase (*this no apunta a la instancia*).
  - Bind en el constructor
  - Usar arrow functions (*public class field syntax*)
  - Arrow functions inline
  - Bind inline

```
// elemento button con manejador de click
<button onClick={this.handleClick}>Click here!</button>
// se ha definido el manejador como método de clase, this.setState es undefined
handleClick() {
  console.log(this.setState);
}
// Bind en el constructor (this apunta a la instancia en el método handleClick)
this.handleClick = this.handleClick.bind(this);
// se define el manejador como arrow function (this apunta a la instancia)
handleClick = () => {
  console.log(this.setState);
}
```

# Listas y keys

- Transformar array en listas de elementos con **Array.map**

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
);
```

- **key:** Atributo especial de los elementos en una lista que ayuda a React a identificar que elementos cambian, son añadidos o borrados. Son obligatorios o React nos lanza un warning.
- keys deben ser únicas para cada elemento en la lista, no se recomienda usar el index si el orden de elementos puede cambiar.
- Solo tienen sentido en el contexto de una lista. NO son props de un componente.

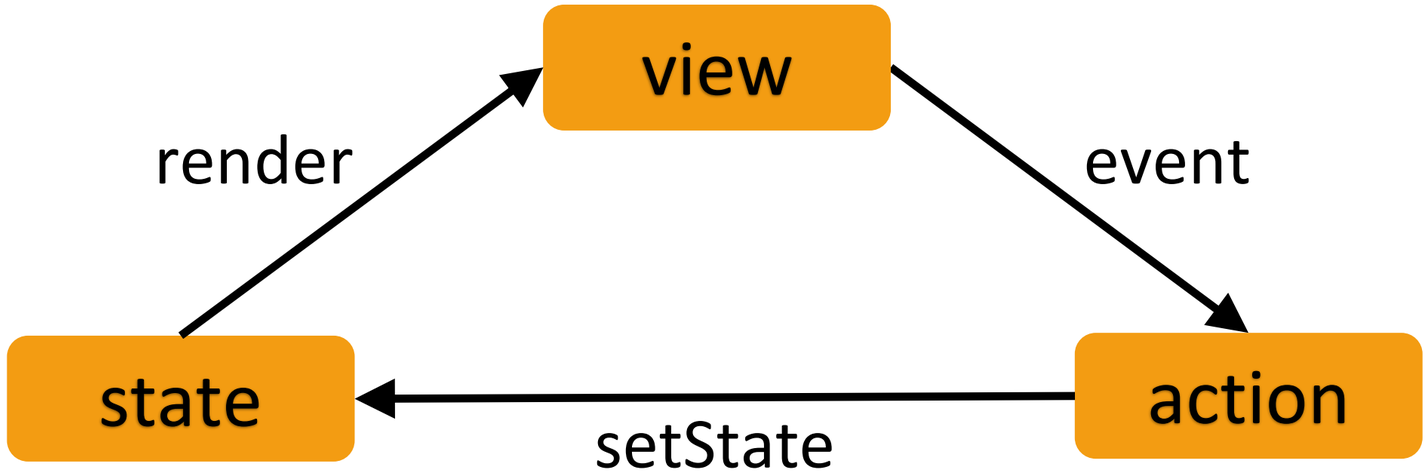
# Forms: componentes controlados

- Mantenemos el valor del elemento input (<input>, <textarea>, <select>) en el **estado de React**, en lugar de mantenerlo en el elemento DOM. **(React = source of truth).**

```
// 1. Definimos un estado en el constructor donde guardar el valor del input
this.state = {email: '', password: ''};

// 2. Definimos un método para el evento change del input, y actualizamos el estado en respuesta al evento
handleChange = (event) => {
  this.setState({value: event.target.value});
}

// 3. Creamos un método para el submit del formulario, donde evitamos el comportamiento por defecto del elemento del DOM
handleSubmit(event) {
  event.preventDefault();
  console.log(this.state.value); // Tenemos acceso al state para hacer una llamanda a una API, p.ej…
}

// 4. Pasamos al input el valor y el método handleChange, y al form el método handleSubmit
<form onSubmit={this.handleSubmit}>
  <label htmlFor="query">
    Name:
    <input name="query" type="text" value={this.state.email} onChange={this.handleChange} />
  </label>
  <input type="submit" value="Submit" />
</form>
```

# Forms: componentes controlados (II)

- **One-way data binding**: La actualización del estado es explícita. La vista llama a las acciones que modifican el estado.




- En ningún caso, la vista actualiza el estado automáticamente (Two-way data binding).

# Forms: componentes controlados (III)

- **<textarea />:** Usamos **value** en lugar de children.
- **<select />:** Usamos **value** en el elemento **select** en lugar de selected en el option y un **value** distinto para cada opción.
- **<input type="checkbox" />**: Usamos **checked** en lugar de value.
- **<input type="radio" />**: Usamos **checked** para el elemento seleccionado y un **value** distinto para cada opción.
- **<input type="file" />**: **Uncontrolled** en react.

- Especificar **null** o **undefined** en un value convierte el input en **uncontrolled (DOM = source of truth)**.

# Forms: componentes no controlados

- Los datos de los inputs son manejados en el DOM, en lugar del estado de React. En general se recomiendan los **controlados**.
- Se usa una **ref** para acceder al valor del input.

```
// 1. Definimos una ref en el constructor para acceder al elemento input
this.input = React.createRef();

// 2. En el submit podemos acceder al valor del input a través de la ref
handleSubmit(event) {
  event.preventDefault();
  console.log(`A name was submitted: ' ${this.input.current.value}`);
}

// 3. pasamos la ref al elemento input
<form onSubmit={this.handleSubmit}>
  <label htmlFor="query">
    Name:
    <input name="query" type="text" ref={this.input} />
  </label>
  <input type="submit" value="Submit" />
</form>
```

- Si necesitamos pasar un valor inicial a un input usamos **defaultValue.**
- Los inputs de tipo **file** son no controlados. (**this.input.current.files**)

# Elevando el estado

- Distintos componentes necesitan compartir estado.
- Situar el estado en un ancestro común.
- Pasar props (datos y funciones) desde el componente que tiene el estado hacia abajo (flujo de props de arriba a abajo). De ese modo los hijos tienen el modo de cambiar el estado en el padre.
- Solo debería haber una única fuente de verdad para un dato.
- Si un dato puede ser calculado a partir del estado o de props, entonces no debería estar en el estado (mantener el estado al mínimo).

# Herramientas: react developer tools

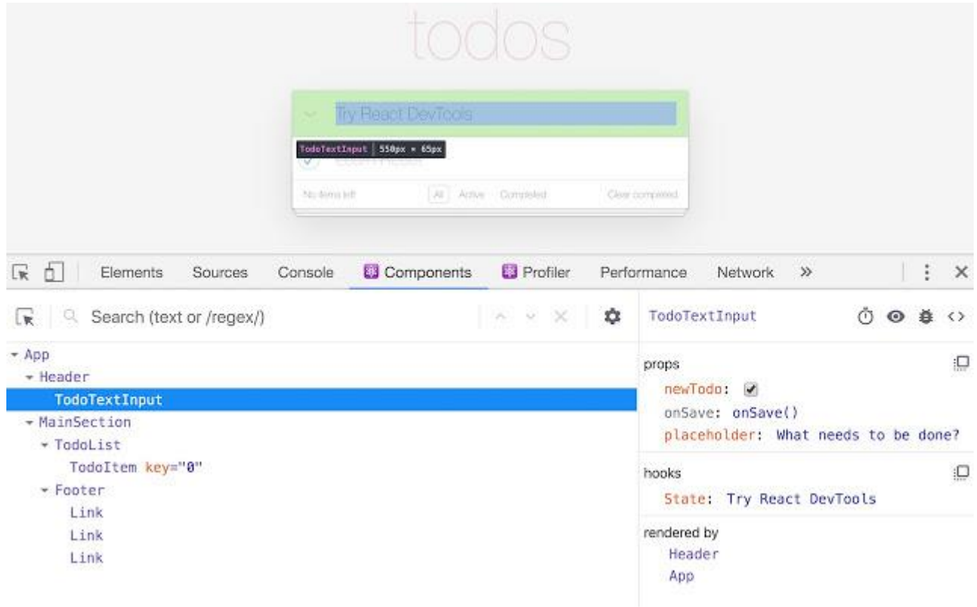
- Con **create-react-app** podemos crear un proyecto React totalmente configurado, sin preocuparnos de webpack, babel, eslint, jest…

```
npx create-react-app my-app
cd my-app
npm start
```

- Scripts disponibles:
  - **npm start**: iniciar en modo development http://localhost:3000
  - **npm test:** ejecutar tests con **Jest**
  - **npm run build:** hacer un build listo para producción
  - **npm run eject:** extraer toda la configuración, **irreversible!!!**
- Más info: https://create-react-app.dev/

# Herramientas: react developer tools

- **Extensión del navegador** que permite inspeccionar la jerarquía de componentes de una aplicación React, sus props, estado….
- Disponible en los principales navegadores




# Aplicando estilos en componentes

- Pasando clases a los elementos mediante el atributo **className.**

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

La librería [classnames](#) nos ayuda a componer clases.

**create-react-app** permite usar **CSS modules, Sass, post-css**

- Pasando un objeto de estilos inline mediante el atributo **style,** con los atributos en *camelCase.*

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

# Aplicando estilos en componentes (II)

- **CSS-in-JS:** CSS compuesto usando Javascript
- Styled Components: CSS con *template literals*

```
const Button = styled.button`
  border-radius: 3px;
  padding: 0.5rem 0;
  margin: 0.5rem 1rem;
  border: 2px solid white;
  ${props => props.primary && css`
    background: white;
    color: black;
  `}
`;
```

- Otras opciones en el ecosistema: Aphrodite, Radium, react-jss, emotion, y muchas más…

# Chequeo dinámico de props: propTypes

- **prop-types:** Paquete donde se definen los distintos propTypes.
  - `npm install --save prop-types`
  - Todos los tipos definidos aqui
  - En Desarrollo React lanza warnings en la consola
- Definimos las prop types de un componente

```
import PropTypes from 'prop-types';
class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}
Greeting.propTypes = {
  name: PropTypes.string
};
```

# Chequeo dinámico de props: propTypes (II)

- Podemos definir propiedades *obligatorias* (requeridas).

```
Greeting.propTypes = {
  name: PropTypes.string.isRequired
};
```

- Podemos definir valores por defecto de las props. Este valor será usado cuando no le pasamos la prop al componente. Este valor por defecto también tiene que pasar el chequeo de propTypes.

```
Greeting.defaultProps = {
  name: 'Stranger'
};
```

# Fragments

- Componente que permite agrupar una listado de *children* sin añadir nodos extras al DOM
  - Resuelven el problema de crear HTML no válido por añadir nodos intermedios
  - Fragments en un array deben llevar *key*

```
// sintaxis normal
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```

```
// sintaxis abreviada
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```

# Refs y el DOM

- Las *refs* nos permiten acceder a hijos y a toda su API desde un componente padre:
  - Elementos del DOM
  - Instancias de componentes
  - No podemos pasar refs a componentes function
- Sugerencias de uso:
  - Manejo de foco, selección de texto, reproducción media
  - Lanzar animaciones imperativamente
  - Integración de librerías de terceros de acceso a DOM
- Pero…**No abusar del uso de refs!!!**

# Refs y el DOM (II)

- Creando una ref y pasándola a un elemento

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    // 1. creando y almacenando la ref
    this.myRef = React.createRef();
  }

  componentDidMount () {
    // 3. accediendo al elemento y a toda su API a través de la ref
    const node =  this.myRef.current;
    node.focus();
  }

  render() {
    // 2. pasando la ref al elemento
    return <input ref={this.myRef} />;
  }
}
```

# Exponiendo refs

- Técnica para exponer la ref de un elemento al exterior de un componente, útil si estamos creando una librería de componentes que se comportan como los elementos nativos HTML.

```
// En lugar de definir el componente así…
const MyButton = props => (
  <button className="my-button">
    {props.children}
  </button>
);
// Lo definimos así
const MyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className=“my-button">
    {props.children}
  </button>
));
// Y podemos obtener la ref al button
const ref = React.createRef();
<MyButton ref={ref}>Click me!</MyButton>;
```

# Contexto: creando el contexto

- El **contexto** nos da un modo de pasar props a través del árbol de componentes sin pasar manualmente las props por cada nivel.
- Está pensado para compartir data cosiderada *global* (usuario autenticado, tema visual, lenguaje preferido…)
- Creamos un objeto de contexto:

```
const MyContext = React.createContext(defaultValue);
```

- **Context.Provider:** Componente que permite a los consumidores subscirbirse a los cambios en el contexto.

```
<MyContext.Provider value={/* some value */}>
```

- La prop **value** es pasada a los componente consumidores descendentes del Provider.

# Contexto: consumiendo el contexto

- **Class.contextType:** Asignamos la propiedad de la clase al objeto de contexto al que nos queremos conectar.

```
class MyClass extends React.Component {
  render() {
    let value = this.context;
    /* renderizamos algo basado en el value del contexto */
  }
}
MyClass.contextType = MyContext;
```

- **Context.Consumer:** Componente que se subscribe a los cambios en el contexto. Acepta una *function as a child* que recibe el value

```
<MyContext.Consumer>
  {value => /* renderizamos algo basado en el value del contexto */}
</MyContext.Consumer>
```