QUICK START

COMMUNITY RESOURCES

GUIDES

REFERENCE

TIPS

CONTRIBUTING

Edit on GitHub

Tutorial

Costruiremo una semplice ma realistica casella dei commenti che puoi inserire in un blog, una versione base dei commenti in tempo reale offerti da Disqus, LiveFyre o Facebook.

Forniremo:

- Una vista di tutti i commenti
- Un modulo per inviare un commento
- Hook perché tu fornisca un backend personalizzato

Avrà anche un paio di caratteristiche interessanti:

- **Commenti ottimistici:** i commenti appaiono nella lista prima che vengano salvati sul server, in modo da sembrare più veloce.
- **Aggiornamenti in tempo reale:** i commenti degli altri utenti sono aggiunti alla vista dei commenti in tempo reale.
- **Formattazione Markdown:** gli utenti possono utilizzare il Markdown per formattare il proprio testo.

Vuoi saltare tutto questo e vedere semplicemente il sorgente?

It's all on GitHub.

Eseguire un server

Per cominciare questo tutorial dobbiamo richiedere un server in esecuzione. Questo servirà puramente come un endpoint per le API che useremo per ottenere e salvare i dati. Per rendere questo compito il più semplice possibile, abbiamo creato un seplice server in un numero di linguaggi di scripting che fa esattamente ciò che ci serve. **Puoi leggere il sorgente o scaricare un file zip contenente tutto ciò che ti serve per cominciare.**

Per semplicità, il server che eseguiremo usa un file `JSON` come database. Non è ciò che eseguiresti in produzione, ma rende più facile simulare ciò che faresti per consumare una API. Non appena avvii il server, supporterà il nostro endpoint API e servirà anche le pagine statiche di cui abbiamo bisogno.

Per Cominciare

Per questo tutorial renderemo il tutto il più semplice possibile. Incluso nel pacchetto del server discusso in precedenza si trova un file HTML su cui lavoreremo. Apri `public/index.html` nel tuo editor preferito. Dovrebbe apparire simile a quanto segue (con qualche piccola differenza, aggiungeremo un tag `<script>` aggiuntivo in seguito):

Code

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
    <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
    <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
    <script src="https://unpkg.com/jquery@3.1.0/dist/jquery.min.js"></script>
    <script src="https://unpkg.com/remarkable@1.7.1/dist/remarkable.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
    <script type="text/babel">
      // To get started with this tutorial running your own code, simply remove
      // the script tag loading scripts/example.js and start writing code here.
    </script>
  </body>
</html>
```

Nel resto di questo tutorial, scriveremo il nostro codice JavaScript in questo tag script. Non disponiamo di alcun aggiornamento in tempo reale avanzato, quindi dovremo aggiornare il browser per vedere gli aggiornamenti dopo il salvataggio. Segui il tuo progresso aprendo `http://localhost:3000` nel tuo browser (dopo aver avviato il server). Quando carichi la pagina per la prima volta senza apportare cambiamenti, vedrai il risultato finale di ciò che ci apprestiamo a costruire. Quando sei pronto per cominciare a lavorare, elimina il tag `<script>` precedente e quindi sei pronto a continuare.

> **Nota:**

> Abbiamo incluso jQuery perché vogliamo semplificare il codice delle nostre chiamate ajax future, ma **NON** è richiesto per far funzionare React.

Il tuo primo componente

In React ciò che importa sono i componenti modulari e componibili. Per il nostro esempio della casella dei commenti, avremo la seguente struttura dei componenti:

Code

```
- CommentBox
  - CommentList
    - Comment
  - CommentForm
```

Costruiamo il componente `CommentBox`, che consiste in un semplice `<div>`:

Code

```
// tutorial1.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        Ciao, mondo! Sono un CommentBox.
      </div>
    );
  }
});
ReactDOM.render(
  <CommentBox />,
  document.getElementById('content')
);
```

Nota che i nomi degli elementi nativi HTML cominciano con una lettera minuscola, mentre i nomi di classe personalizzati di React cominciano con una lettera maiuscola.

Sintassi JSX

La prima cosa che noterai è la sintassi simile a XML nel tuo JavaScript. Abbiamo un semplice preprocessore che traduce lo zucchero sintattico a questo semplice JavaScript:

Code

```
// tutorial1-raw.js
var CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Ciao, mondo! Sono un CommentBox."
      )
    );
  }
});
ReactDOM.render(
```

```
    React.createElement(CommentBox, null),
    document.getElementById('content')
  );
```

Il suo uso è opzionale ma abbiamo trovato la sintassi JSX più facile da usare del semplice JavaScript. Leggi maggiori dettagli sull'articolo sulla sintassi JSX.

Cosa sta succedendo?

Passiamo dei metodi in un oggetto JavaScript a `React.createClass()` per creare un nuovo componente React. Il più importante di questi metodi è chiamato `render` il quale restituisce un albero di componenti React che saranno eventualmente visualizzati come HTML.

I tag `<div>` non sono veri nodi DOM; sono istanze dei componenti React `div`. Puoi pensare a questi come marcatori o a elementi di dati che React sa come gestire. React è **sicuro**. Non stiamo generando stringhe HTML quindi la protezione XSS è predefinita.

Non devi necessariamente restituire semplice HTML. Puoi anche restituire un albero di componenti costruiti da te (o da qualcun altro). Questo è ciò che rende React **componibile**: una caratteristica chiave dei front-end manutenibili.

`ReactDOM.render()` istanzia il componente radice, avvia il framework, e inietta il markup in un elemento DOM nativo, fornito come secondo argomento.

Comporre componenti

Costruiamo degli scheletri per `CommentList` e `CommentForm` che saranno, nuovamente, dei semplici `<div>`. Aggiungi questi due componenti al tuo file, mantenendo la dichiarazione esistente di `CommentBox` e la chiamata a `ReactDOM.render`:

Code

```
// tutorial2.js
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = React.createClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

Successivamente, aggiorna il componente `CommentBox` per utilizzare questi due nuovi componenti:

Code

```
// tutorial3.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Commenti</h1>
        <CommentList />
        <CommentForm />
      </div>
    );
  }
});
```

Nota come stiamo mescolando tag HTML e i componenti che abbiamo costruito. I componenti HTML sono componenti regolari React, proprio come quelli che definisci, con una differenza. Il compilatore JSX riscriverà automaticamente i tag HTML come espressioni `React.createElement(tagName)` e lascerà tutto il resto inalterato. Questo è per impedire che il namespace globale venga inquinato.

Usare le proprietà

Creiamo il componente `Comment`, che dipenderà dai dati passatigli dal suo genitore. I dati passati da un componente genitore sono disponibili come una 'proprietà' nel componente figlio. Queste 'proprietà' sono accessibili attraverso `this.props`. Usando le proprietà, saremo in grado di leggere i dati passati al componente `Comment` dal componente `CommentList`, e visualizzare del markup:

Code

```
// tutorial4.js
var Comment = React.createClass({
  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {this.props.children}
      </div>
    );
  }
});
```

Racchiudendo un'espressione JavaScript dentro parentesi graffe all'interno di JSX (sia come attributo che come figlio), puoi inserire del testo o componenti React all'interno dell'albero. Accediamo per nome ad attributi passati al componente tramite chiavi in `this.props` e ciascun elemento annidato come `this.props.children`.

Proprietà dei Componenti

Adesso che abbiamo definito il componente `Comment`, vogliamo passargli il nome dell'autore e il testo del commento. Questo ci permette di riutilizzare lo stesso codice per ciascun commento individuale. Ora aggiungiamo dei commenti all'interno del nostro `CommentList`:

Code

```
// tutorial5.js
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        <Comment author="Pete Hunt">Questo è un commento</Comment>
        <Comment author="Jordan Walke">Questo è un *altro* commento</Comment>
      </div>
    );
  }
});
```

Nota che abbiamo passato dei dati dal componente genitore `CommentList` al componente figlio `Comment`. Ad esempio, abbiamo passato *Pete Hunt* (tramite un attributo) e *Questo è un commento* (tramite un nodo figlio simil-XML) al primo `Comment`. Come detto in precedenza, il componente `Comment` accederà a queste 'proprietà' attraverso `this.props.author` e `this.props.children`.

Aggiungiamo il Markdown

Il Markdown è una maniera semplice di formattare il tuo testo in linea. Ad esempio, racchiudendo il testo con asterischi gli aggiungerà dell'enfasi.

Per prima cosa, aggiungiamo la libreria di terze parti **marked** alla tua applicazione. Questa è una libreria JavaScript che prende il testo Markdown e lo converte in HTML nativo. Per fare ciò è richiesto un tag script nel tag head (che abbiamo già incluso nel playground React):

Code

```
<!-- index.html -->
<head>
  <meta charset="utf-8" />
  <title>React Tutorial</title>
  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
  <script src="https://unpkg.com/jquery@3.1.0/dist/jquery.min.js"></script>
  <script src="https://unpkg.com/remarkable@1.7.1/dist/remarkable.min.js"></script>
</head>
```

Successivamente, convertiamo il testo del commento da Markdown e scriviamolo:

Code

```
// tutorial6.js
var Comment = React.createClass({
  render: function() {
    var md = new Remarkable();
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {md.render(this.props.children.toString())}
```

```
        </div>
      );
    }
  });
```

Tutto ciò che stiamo facendo qui è chiamare la libreria remarkable. Dobbiamo convertire `this.props.children` dal testo racchiuso di React a una stringa che remarkable è in grado di capire, quindi vi chiamiamo esplicitamente `toString()`.

Ma c'è un problema! I nostri commenti visualizzati appaiono come segue nel browser: " `<p>` Questo è un `<em>` altro `</em>` commento `</p>` ". Noi vogliamo che questi tag vengano in realtà visualizzati come HTML.

Questo è il risultato della protezione di React da parte di un attacco XSS. C'è una maniera di aggirare questo comportamento, ma il framework ti avvisa di non farlo:

Code

```
// tutorial7.js
var Comment = React.createClass({
  rawMarkup: function() {
    var md = new Remarkable();
    var rawMarkup = md.render(this.props.children.toString());
    return { __html: rawMarkup };
  },

  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        <span dangerouslySetInnerHTML={this.rawMarkup()} />
      </div>
    );
  }
});
```

Questa è una speciale API che rende intenzionalmente difficile inserire HTML nativo, ma per remarkable ci avvantaggeremo di questa possibilità.

Ricorda: usando questa funzionalità stai assumendo che remarkable sia sicuro.

Collega il modello dei dati

Finora abbiamo inserito i commenti direttamente nel codice sorgente. Invece, visualizziamo un pacchetto di dati JSON nella lista dei commenti. In seguito questi verranno restituiti dal server, ma per adesso scriviamoli nel tuo sorgente:

Code

```
// tutorial8.js
var data = [
  {author: "Pete Hunt", text: "Questo è un commento"},
  {author: "Jordan Walke", text: "Questo è un *altro* commento"}
];
```

Dobbiamo inserire questi dati in `CommentList` in maniera modulare. Modifica `CommentBox` e la chiamata a `ReactDOM.render()` per passare questi dati a `CommentList` tramite proprietà:

Code

```
// tutorial9.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Commenti</h1>
        <CommentList data={this.props.data} />
        <CommentForm />
      </div>
    );
  }
});

ReactDOM.render(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

Adesso che i dati sono disponibili in `CommentList`, visualizziamo i commenti dinamicamente:

Code

```
// tutorial10.js
var CommentList = React.createClass({
  render: function() {
    var commentNodes = this.props.data.map(function (comment) {
      return (
        <Comment author={comment.author} key={comment.id}>
          {comment.text}
        </Comment>
      );
    });
    return (
      <div className="commentList">
        {commentNodes}
      </div>
    );
  }
});
```

Tutto qui!

Richiedere dati dal server

Sostituiamo i dati scritti nel codice con dati dinamici ottenuti dal server. Rimuoveremo le proprietà dei dati e le sostituiremo con uno URL da richiedere:

Code

```
// tutorial11.js
ReactDOM.render(
  <CommentBox url="/api/comments" />,
  document.getElementById('content')
);
```

Questo componente differisce dal precedente perché dovrà effettuare un nuovo rendering di se stesso. Il componente non avrà dati finché la risposta del server non sia disponibile, e a quel punto il componente potrebbe dover visualizzare dei nuovi commenti.

Nota: il codice non funzionerà a questo passo.

Stato reattivo

Finora, basandosi sulle sue proprietà, ogni componente si è visualizzato solo una volta. I valori di `props` sono immutabili: sono passati dal genitore e sono "posseduti" dal genitore. Per implementare le interazioni, introduciamo uno **stato** mutevole nel componente. `this.state` è privato al componente e può essere cambiato chiamando `this.setState()`. Quando lo stato si aggiorna, il componente effettua nuovamente il rendering di se stesso.

I metodi `render()` sono scritti dichiarativamente come funzioni di `this.props` e `this.state`. Il framework garantisce che la UI sia sempre consistente con gli input.

Quando il server ci fornisce i dati, dovremo cambiare i dati dei commenti in nostro possesso. Aggiungiamo un array di dati dei commenti al componente `CommentBox` come il suo stato:

Code

```
// tutorial12.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Commenti</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});
```

`getInitialState()` viene eseguito esattamente una volta durante il ciclo di vita del componente e imposta lo stato iniziale del componente stesso.

Aggiornare lo stato

Quando il componente è creato per la prima volta, vogliamo richiedere tramite GET del JSON dal server e aggiornare lo stato per riflettere i dati più recenti. Useremo jQuery per effettuare una richiesta asincrona al server che abbiamo avviato in precedenza per richiedere i dati che ci servono. Somiglieranno a qualcosa di simile:

Code

```
[
  {"author": "Pete Hunt", "text": "Questo è un commento"},
  {"author": "Jordan Walke", "text": "Questo è un *altro* commento"}
]
```

Code

```
// tutorial13.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Commenti</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});
```

Qui, `componentDidMount` è un metodo chiamato automaticamente da React quando un componente viene visualizzato. La chiave agli aggiornamenti dinamici è la chiamata a `this.setState()`. Sostituiamo il vecchio array di commenti con il nuovo ottenuto dal server e la UI si aggiorna automaticamente. Per via di questa reattività, è richiesto soltanto un piccolo cambiamento per aggiungere gli aggiornamenti in tempo reale. Qui useremo un semplice polling, ma potrai facilmente usare WebSockets o altre tecnologie.

Code

```
// tutorial14.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
```

```
        error: function(xhr, status, err) {
          console.error(this.props.url, status, err.toString());
        }.bind(this)
      });
    },
    getInitialState: function() {
      return {data: []};
    },
    componentDidMount: function() {
      this.loadCommentsFromServer();
      setInterval(this.loadCommentsFromServer, this.props.pollInterval);
    },
    render: function() {
      return (
        <div className="commentBox">
          <h1>Commenti</h1>
          <CommentList data={this.state.data} />
          <CommentForm />
        </div>
      );
    }
  });

  ReactDOM.render(
    <CommentBox url="/api/comments" pollInterval={2000} />,
    document.getElementById('content')
  );
```

Tutto ciò che abbiamo fatto finora è spostare la chiamata AJAX in un metodo a parte e chiamarlo quando il componente viene caricato per la prima volta e successivamente ogni 2 secondi. Prova ad eseguire questa versione nel tuo browser e a cambiare il file `comments.json` (si trova nella stessa directory del tuo server); entro 2 secondi i cambiamenti saranno visibili!

Aggiungere nuovi commenti

È giunto il momento di costruire il modulo. Il nostro componente `CommentForm` deve chiedere all'utente il nome e un testo del commento, e inviare una richiesta al server per salvare il commento.

Code

```
// tutorial15.js
var CommentForm = React.createClass({
  render: function() {
    return (
      <form className="commentForm">
        <input type="text" placeholder="Il tuo nome" />
        <input type="text" placeholder="Di' qualcosa..." />
        <input type="submit" value="Invia" />
      </form>
    );
  }
});
```

Rendiamo il modulo interattivo. Quando l'utente invia il modulo, dobbiamo ripulirlo, inviare una richiesta al server, e aggiornare la lista dei commenti. Per cominciare, ascoltiamo l'evento `submit` del modulo e ripuliamolo.

Code

```
// tutorial16.js
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var author = ReactDOM.findDOMNode(this.refs.author).value.trim();
    var text = ReactDOM.findDOMNode(this.refs.text).value.trim();
    if (!text || !author) {
      return;
    }
    // TODO: invia la richiesta al server
    ReactDOM.findDOMNode(this.refs.author).value = '';
    ReactDOM.findDOMNode(this.refs.text).value = '';
    return;
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input type="text" placeholder="Il tuo nome" ref="author" />
        <input type="text" placeholder="Di' qualcosa..." ref="text" />
        <input type="submit" value="Invia" />
      </form>
    );
  }
});
```

Eventi

React assegna i gestori degli eventi ai componenti usando una convenzione di nomi camelCased. Assegnamo un gestore `onSubmit` al modulo in maniera che ripulisca i campi del modulo quando il modulo stesso è inviato con un input valido.

Chiamiamo `preventDefault()` sull'evento per prevenire l'azione predefinita del browser per l'invio del modulo.

Refs

Usiamo l'attributo `ref` per assegnare un nome a un componente figlio e `this.refs` per riferirsi al componente. Possiamo chiamare `ReactDOM.findDOMNode(component)` su di un componente per ottenere l'elemento nativo del DOM del browser.

Callback come proprietà

Quando un utente invia un commento, dobbiamo aggiornare la lista dei commenti per includere il nuovo commento. Ha senso posizionare questa logica in `CommentBox` dal momento che `CommentBox` possiede lo stato che rappresenta la lista dei commenti.

Dobbiamo passare i dati dal componente figlio su fino al suo genitore. Lo facciamo nel metodo `render` del nostro genitore passando una nuova callback (`handleCommentSubmit`) al figlio, legandola all'evento `onCommentSubmit` del figlio. Quando questo evento viene emesso, la callback verrà eseguita:

Code

```
// tutorial17.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    // TODO: invia al server e aggiorna la lista
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Commenti</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

Chiamiamo la callback da `CommentForm` quando l'utente invia il modulo:

Code

```
// tutorial18.js
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var author = ReactDOM.findDOMNode(this.refs.author).value.trim();
    var text = ReactDOM.findDOMNode(this.refs.text).value.trim();
    if (!text || !author) {
      return;
    }
    this.props.onCommentSubmit({author: author, text: text});
    ReactDOM.findDOMNode(this.refs.author).value = '';
    ReactDOM.findDOMNode(this.refs.text).value = '';
    return;
  },
```

```
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input type="text" placeholder="Il tuo nome" ref="author" />
        <input type="text" placeholder="Di' qualcosa..." ref="text" />
        <input type="submit" value="Invia" />
      </form>
    );
  }
});
```

Adesso che le callback sono al loro posto, non ci resta che inviare al server e aggiornare la lista:

Code

```
// tutorial19.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Commenti</h1>
```

```
          <CommentList data={this.state.data} />
          <CommentForm onCommentSubmit={this.handleCommentSubmit} />
        </div>
      );
    }
  });
```

Ottimizzazione: aggiornamenti ottimistici

La nostra applicazione è adesso completa, ma aspettare che la richiesta completi prima di vedere il commento apparire nella lista la fa sembrare lenta. Possiamo aggiungere ottimisticamente questo commento alla lista per fare apparire l'applicazione più veloce.

Code

```
// tutorial20.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    var comments = this.state.data;
    var newComments = comments.concat([comment]);
    this.setState({data: newComments});
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        this.setState({data: comments});
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
```

```
    },
    render: function() {
      return (
        <div className="commentBox">
          <h1>Commenti</h1>
          <CommentList data={this.state.data} />
          <CommentForm onCommentSubmit={this.handleCommentSubmit} />
        </div>
      );
    }
  });
```

Congratulazioni!

Hai appena costruito una casella di commenti in pochi semplici passi. Leggi maggiori dettagli sul perché usare React, o approfondisci la guida di riferimento dell'API e comincia ad hackerare! In bocca al lupo!

← Prev Next →

A Facebook & Instagram collaboration.
Acknowledgements