# Klasne komponente

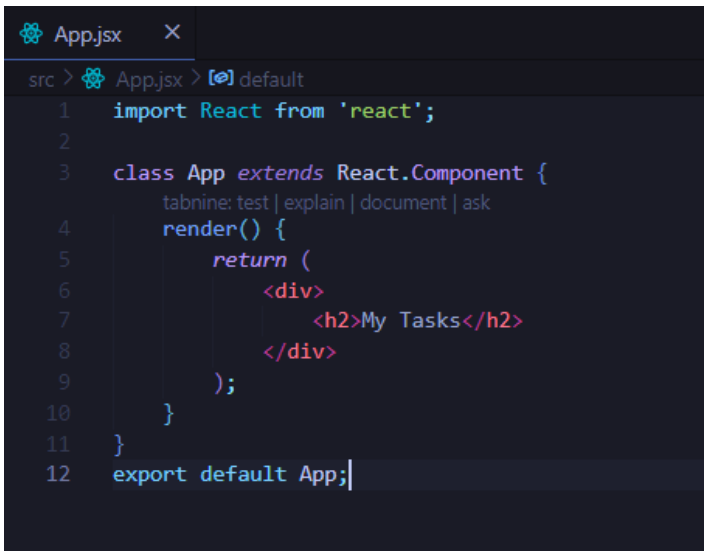
## Šta su klasne komponente?

Kao što smo pomenuli ranije tokom kursa, funkcionalne komponente nisu jedini način da se naprave React komponente. Nekadašnji standard za izradu React.js komponenti su bile takozvane **klasne komponente**. Koristile su se za implementaciju state-a i side effects-a pre React verzije 16.8, odnosno, pre uvođenja React Hooks. U nastavku ćemo naučiti kako da napravimo klasnu komponentu, kako da iskoristimo njene props, kako da napravimo state, kako da modifikujemo state, i najzad, kako da implementiramo side effects.

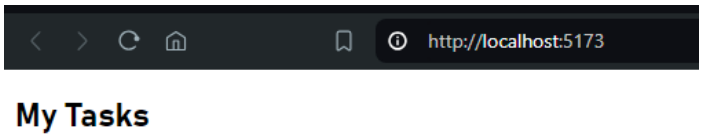
## Definisanje klasne komponente

U React-u, klasne komponente se definišu kao ES6+ kase koje nasleđuju klasu React.Component. Bez ovog nasleđivanja, naša klasa ne bi mogla da prihvata props, upravlja state-om ili lifecycle metodima.

Kada želimo da renderujemo sadržaj JSX-a, i kod klasnih komponenti koristimo **return**, samo što je taj return smešten u okviru **render()** metode. U nastavku ćemo videti primer jedne jednostavne klasne komponente koja na ekranu prikazuje tekst *My Tasks*.




Rezutlat izvršenja prethodnog koda:

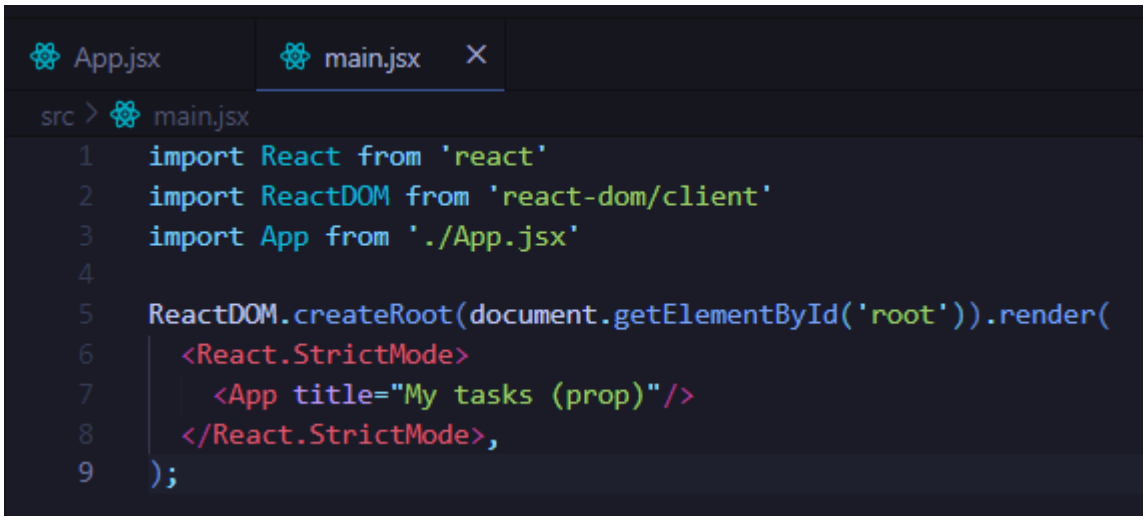



## Prihvatanje props

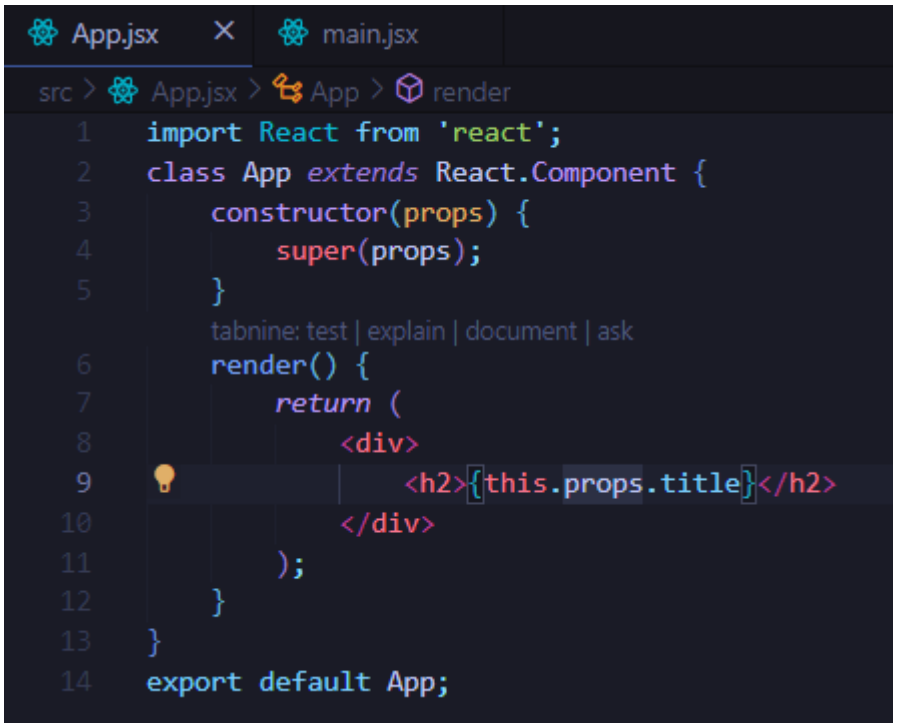
Ukoliko želimo da naša klasna komponenta prihvati props, to možemo uraditi tako što ćemo definisati konstruktor koji prihvata parametar props, a zatim taj parametar kao argument proslediti konstuktoru roditeljske komponente React.Component, koristeći **super(props)**. Kada želimo da se u kodu služimo nekim članom objekta props, to radimo na sledeći način: **this.props.nazivČlana**.

U nastavku ćemo videti modifikovanu verziju prethodne komponente, koja će prihvatiti prop **title,** a zatim prikazati tu vrednost na ekranu.

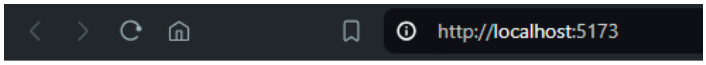
Na prvoj slici možemo videti prosleđivanje prop-a title komponenti App.




Na sledećoj slici možemo videti prihvatanje i korišćenje datog prop-a.




Konačno, na sledećoj slici možemo videti kako naša trenutna komponenta u pretraživaču.




My tasks (prop)

## Formiranje i ažuriranje state-a

Definisanje state-a naše komponente se vrši u konstruktoru. Pristupamo objektu state, članu naše klase, i postavljamo početne vrednosti njegovih članova.

Modifikaciju state-a vršimo u okviru različitih metoda naše klase, kako predefinisanih, tako i definisanih od strane nas. **Jedini dozvoljen način za modifikaciju state-a u okviru klasa je korišćenjem sledeće naredbe: this.setState().** U setState prosleđujemo objekat, koji kao članove sadrži članove našeg state-a koje želimo da modifikujemo, a kao vrednosti ima nove vrednosti.

Pristup članu state-a je omogućen primenom sledećeg koda **this.state.nazivOdgovarajućegČlanaStateObjekta**. Modifikacija state-a na način prikazan u nastavku je **strogo zabranjen**: **this.state.nazivOdgovarajućegČlanaStateObjekta = novaVrednost;**
**Ispravan način: this.setState({nazivOdgovarajućegČlanaStateObjekta: novaVrednost});**

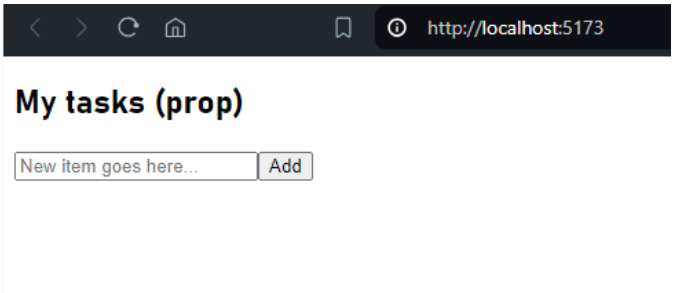
U nastavku sledi primer koda koji predstavlja jednostavnu implementaciju to-do liste. Naša komponenta će imati mogućnost unosa novih zadataka, prikazivanja do sada unetih zadataka, kao i njihovog brisanja.

App.jsx main.jsx

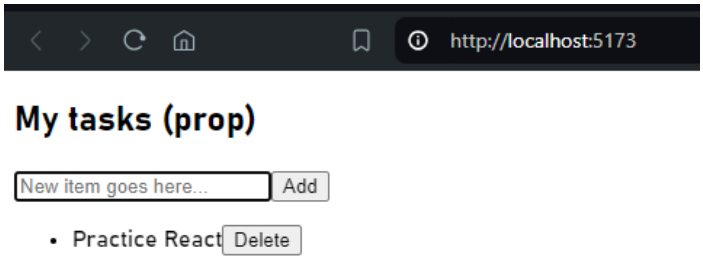
src > App.jsx > App > render

```jsx
import React from 'react';
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = { // definisanje state-a naše komponente
            itemsList: [],
            newItem: '',
        };
    }
    handleChange = (event) => {
        this.setState({ newItem: event.target.value }); // modifikacija newItem člana state-a
    };
    handleSubmit = (event) => {
        event.preventDefault();
        if (this.state.newItem.trim() === '') return;
        const newItemsList = [...this.state.itemsList, { id: crypto.randomUUID(), text: this.state.newItem }];
        this.setState({ itemsList: newItemsList, newItem: '' }); // modifikacija oba člana state-a
    };
    handleDelete = (itemId) => {
        const newItemsList = this.state.itemsList.filter((item) => item.id !== itemId);
        this.setState({ itemsList: newItemsList }); // modifikacija itemsList člana state-a
    };
    tabnine: test | explain | document | ask
    render() {
        return (
            <div>
                <h2>{this.props.title}</h2>
                <form onSubmit={this.handleSubmit}>
                    <input type="text" value={this.state.newItem} // pristup newItem članu state-a
                        onChange={this.handleChange}
                        placeholder="New item goes here..."
                    />
                    <button type="submit">Add</button>
                </form>
                <ul>
                    {
                        this.state.itemsList.map((item) => (
                            <li key={item.id}>
                                {item.text}
                                <button onClick={() => this.handleDelete(item.id)}>Delete</button>
                            </li>
                        ))
                    }
                </ul>
            </div>
        );
    }
}
export default App;
```

U nastavku sledi rezultat prethodnog koda pri prvom renderu komponente.




U nastavku sledi rezultat prethodnog koda nakon unosa item-a *Practice React.*




Nakon pritiska na dugme **Delete**, stranica će izgledati kao na slici pre prethodne.

## Lifecycle metodi

U okviru snimljenih materijala smo se upoznali sa **useEffect** React hook-om, ali, React Hooks nisu dostupni kada koristimo klasne komponente. U okviru klasnih komponenti postoje takozvane lifecycle metode.

Tokom mounting faze, redom se pozivaju sledeće lifecycle metode:

1. **constructor()**
    - Uglavnom se u cilju inicijalizacije state-a.
    - Poziva se kada je komponenta inicijalizovana, ali pre nego što je renderovana.
2. **static getDerivedStateFromProps()**
    - Poziva se nakon inicijalizacije komponente.
    - Dozvoljava komponenti da ažurira state na osnovu promena props objekta.
    - Retko se koristi, i može da izazove puno različitih grešaka.
    - Prihvata dva parametra state i props.
    - Ukoliko je bilo promene, vraća objekat, ukoliko nema potrebe za promenom, vraća null.
3. **render()**
    - Kao što smo mogli da vidimo u prvom primeru koda, ovo je jedini metod koji moramo da iskoristimo kada pravimo klasnu komponentu.
    - Služi za renderovanje sadržaja našeg JSX-a.
4. **componentDidMount()**
    - Poziva se neposredno nakon prvog rendera komponente.
    - Ovo je dobro mesto za ostvarivanje komunikacije sa nekim API-em i realizovanje drugih side effects.

U nastavku sledi kod kojim se prikazuje redosled pozivanja lifecycle funkcija u mounting fazi.

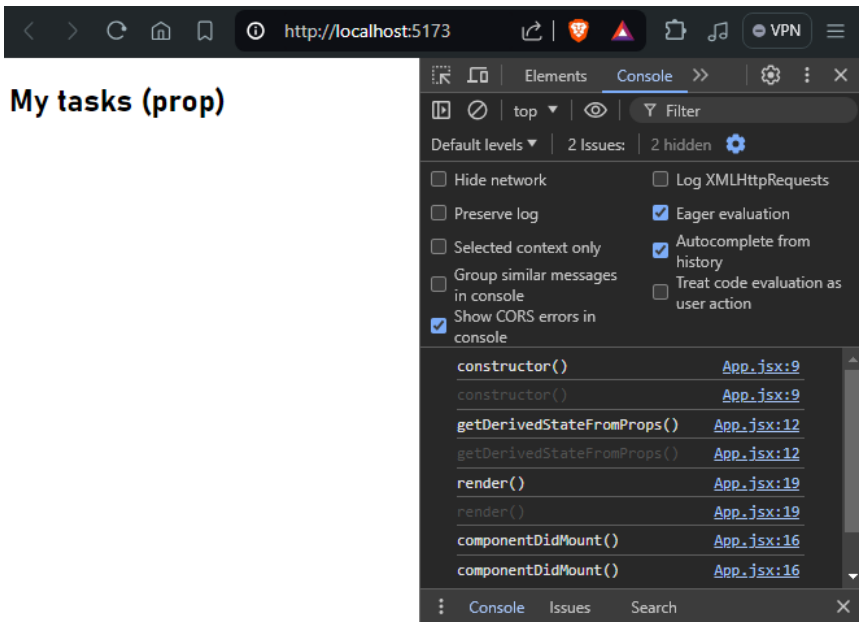
App.jsx

src > App.jsx > App > componentDidMount

```jsx
import React from 'react';
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            itemsList: [],
            newItem: '',
        };
        console.log('constructor()');
    }
    tabnine: test | explain | document | ask
    static getDerivedStateFromProps(props, state) {
        console.log('getDerivedStateFromProps()');
        return null;
    }
    tabnine: test | explain | document | ask
    componentDidMount() {
        console.log('componentDidMount()');
    }
    tabnine: test | explain | document | ask
    render() {
        console.log('render()');
        return (
            <div>
                <h2>{this.props.title}</h2>

            </div>
        );
    }
}
export default App;
```

U nastavku sledi rezultat koda sa prethodne slike.




Tokom updating faze, redom se pozivaju sledeće lifecycle metode:

1. **static getDerivedStateFromProps()**

    - Ukoliko dođe do promene props objekta.
2. **shouldComponentUpdate()**
    - Koristi se za optimizaciju performansi.
    - Vraća true ili false, i vrednošću koju vrati se određuje da li bi komponenta trebalo da se ažurira zbog promena props ili state objekta.
    - Podrazumevano ponašanje je da svaka promena props i state objekta dovodi do ažuriranja i ponovnog renderovanja.
    - Ova metoda kao parametre uzima nextProps i nextState, zbog čega se u okviru ove metode lako mogu uporediti trenutne vrednosti ovih objekata sa njihovim vrednostima nakon promene.
3. **getSnapshotBeforeUpdate()**
    - Poziva se neposredno pre nego što se promene ovog ažuriranja odraze na DOM.
    - Rezultat ove metode se prosleđuje kao treći parametar componentDidUpdate() metode.
4. **render()**
    - Zarad renderovanja JSX-a modifikovanog u skladu sa promenama.
5. **componentDidUpdate()**
    - Kao parametar prihvata prethodno stanje props objekta, prethodno stanje state objekta i povratnu vrednost getSnapshotBeforeUpdate() metode, ukoliko postoji u kodu.
    - Često se koristi za pokretanje različitih side effects u zavisnosti od razlike između prethodnih i trenutnih vrednosti state i props objekta.

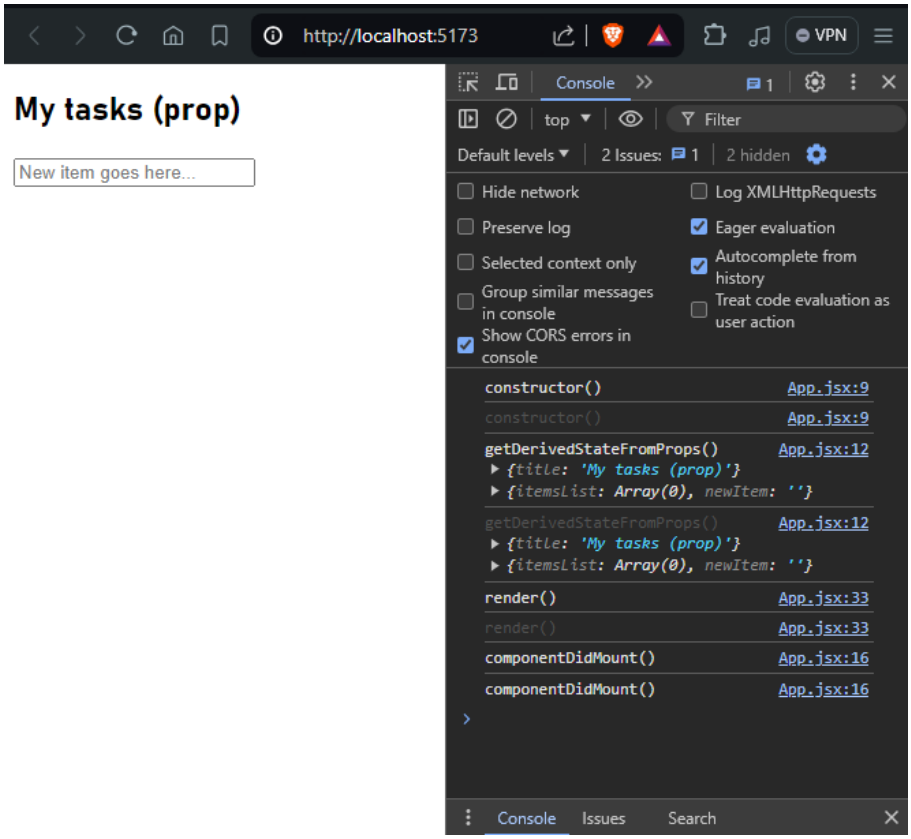
U nastavku sledi pojednostavljena verzija komponente koju smo napravili u okviru **Formiranje i ažuriranje state-a** sekcije, proširena lifecycle metodama koje smo do sada predstavili.

App.jsx

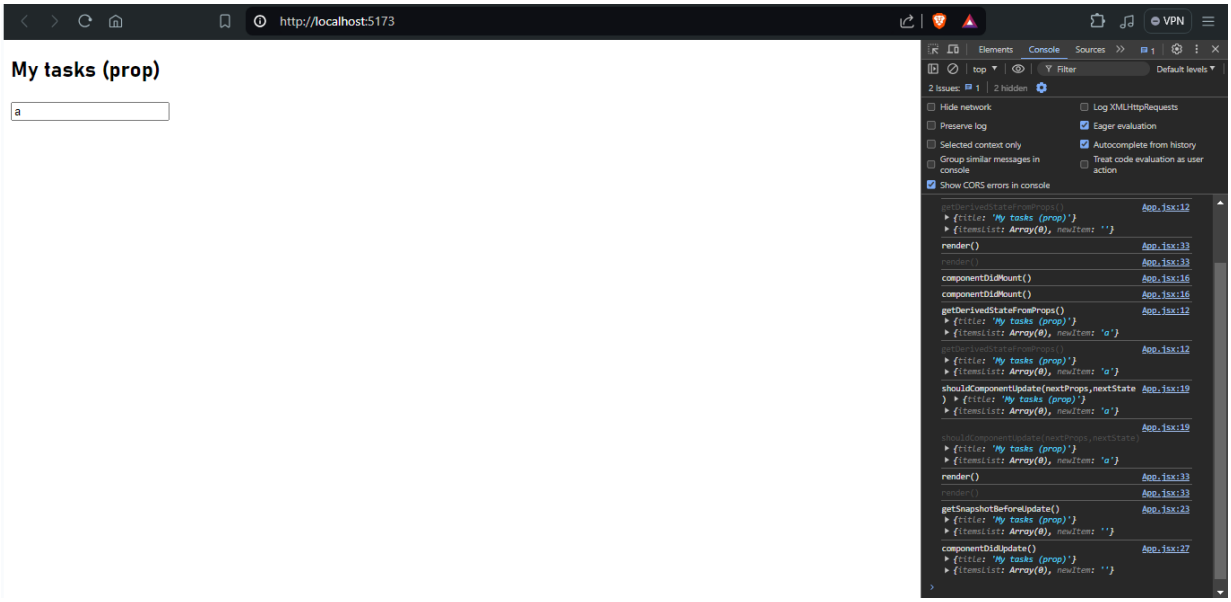
src > App.jsx > App > getSnapshotBeforeUpdate

```jsx
import React from 'react';
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            itemsList: [],
            newItem: '',
        };
        console.log('constructor()');
    }
    tabnine: test | explain | document | ask
    static getDerivedStateFromProps(props, state) {
        console.log('getDerivedStateFromProps()', props, state);
        return null;
    }
    tabnine: test | explain | document | ask
    componentDidMount() {
        console.log('componentDidMount()');
    }
    tabnine: test | explain | document | ask
    shouldComponentUpdate(nextProps, nextState) {
        console.log('shouldComponentUpdate(nextProps,nextState)', nextProps, nextState);
        return true;
    }
    tabnine: test | explain | document | ask
    getSnapshotBeforeUpdate(prevProps, prevState) {
        console.log('getSnapshotBeforeUpdate()', prevProps, prevState);
        return null;
    }
    tabnine: test | explain | document | ask
    componentDidUpdate(prevProps, prevState) {
        console.log('componentDidUpdate()', prevProps, prevState);
    }
    handleChange = (event) => {
        this.setState({ newItem: event.target.value });
    };
    tabnine: test | explain | document | ask
    render() {
        console.log('render()');
        return (
            <div>
                <h2>{this.props.title}</h2>
                <input type="text"
                    value={this.state.newItem}
                    onChange={this.handleChange}
                    placeholder="New item goes here..."
                />
            </div>
        );
    }
}
export default App;
```

Rezultat izvršenja prethodnog koda:




Rezultat izvršenja prethodnog koda nakon unosa teksta u input polje (i samim tim modifikacije newItem člana state objekta):




Uveličajmo konzolu, i obratimo pažnju na njen sadržaj nakon ispisa *componentDidMount()*, s obzirom na to da je to sadržaj nastao tokom updating faze.


Tokom unmounting faze poziva se lifecycle metoda componentWillUnmount(). Poziva se pre nego što se komponenta ukloni iz DOM-a, i u okviru nje se izvode "čišćenja" u vidu uklanjanja event listener-a, čišćenja podataka vezanih za ovu komponentu smeštenih u localStorage-u, poništavanja tajmera i drugih.