REACT + ASYNC

PROMISES

The Promise object is used for asynchronous computations. A Promise represents a value which **may be available now, or in the future, or never.**

Ref: MDN - Promises

fetch() *allows you to make network requests similar to* **XMLHttpRequest** *. The main difference is that the Fetch API uses* Promises *, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest.*

Ref: Google Dev: Introduction to fetch()

Ett Promise har 3 states

- pending : initial state, not fulfilled or rejected.
- fulfilled : meaning that the operation completed successfully.
- rejected : meaning that the operation failed.

Ref: MDN - Promises

```
$.ajax({
  url: "https://url-example.com",
  success: function(response){
    console.log(response);
  },
  error: function(err) {
    console.log(error);
  }
})
```

Hanterar både lyckad request samt misslyckad

Alternativ syntax

```
$.get("https://url-example.com")
    .done(function() {
        console.log("OKELDOKELI");
    })
    .fail(function() {
        console.log("WRONG! JUST WRONG!");
    });
```

`.done` och `.fail` "lyssnar" på vad som händer med vår AJAX-request

Promises är `thenable`

Den moderna tidens callback

Ett Promise följs alltid av ett `.then`

```
fetch('https://get.com') //a promise!
  .then(function() {
    console.log('OKELIDOKELI');
  });
```

`then` tar alltid en funktion som argument

```
fetch('https://get.com') //a promise!
  .then(function() {
    console.log('OKELIDOKELI');
  })
  .catch(function(){
    console.log('Oh no..')
  });
```

Det är bra att alltid fånga upp error med `.catch()`

Det som returneras från fetch är alltid ett promise

Vi måste plocka ut vår JSON

`JSON.parse()` eller den kortare `.json()`

```
fetch("https://example.com")          //Promise
  .then(response => response.json())  //Promise
  .then(data => console.log(data));   //json to process
```

`return` här är implicit med arrow function, vi behöver inte skriva ut det

Fat version

```
fetch("https://example.com")
  .then(function(response){
    return response.json()
  })
  .then(function(data){
    console.log(data)
  });
```

data är det som returneras från första `.then()`

OCH I REACT?

Informationen som hämtas och applikationen ska uppdateras när hämtningen är klar

Ändringar i **state** triggar alltid en uppdatering på sidan

State blir vår förvaring, vår låda att lägga alla variabler i.

```
const Module = function(){
  let state = {
    data: []     //"global" state
  }




}
```

```
const Module = function(){
  let state = {
    data: []    //"global" state
  }
  const getDataFromApi = function(){




  }
}
```

```
const Module = function(){
  let state = {
    data: []    //"global" state
  }
  const getDataFromApi = function(){
    fetch("https://example.com")
    .then(response => response.json())
    .then(json => {


    })
  }
}
```

```
const Module = function(){
  let state = {
    data: []    //"global" state
  }
  const getDataFromApi = function(){
    fetch("https://example.com")
    .then(response => response.json())
    .then(json => {
      state.data = json;  //set global state
    })
  }
}
```

```
class App extends Component {
  state = {
    data: [] //"global state"
  }




}
```

```
class App extends Component {
  state = {
    data: [] //"global state"
  }
  getDataFromApi(){
    fetch("https://example.com")
    .then(response => response.json())
    .then(json => {


    })
  }
}
```

```
class App extends Component {
  state = {
    data: [] //"global state"
  }
  getDataFromApi(){
    fetch("https://example.com")
    .then(response => response.json())
    .then(json => {
      this.setState({data : json});  //set global state
    })
  }
}
```

Utifrån state får vi sedan rendera ut vårt innehåll

```
//App.js
render(){
  const list = this.state.data.map(item =>
    <li> { item.name } </li>
  );
  return (
    <div>
      { list }
    </div>
  )
}
```

COMPONENT LIFECYCLE

```
import React, { Component } from 'react';
class App extends Component{}
```

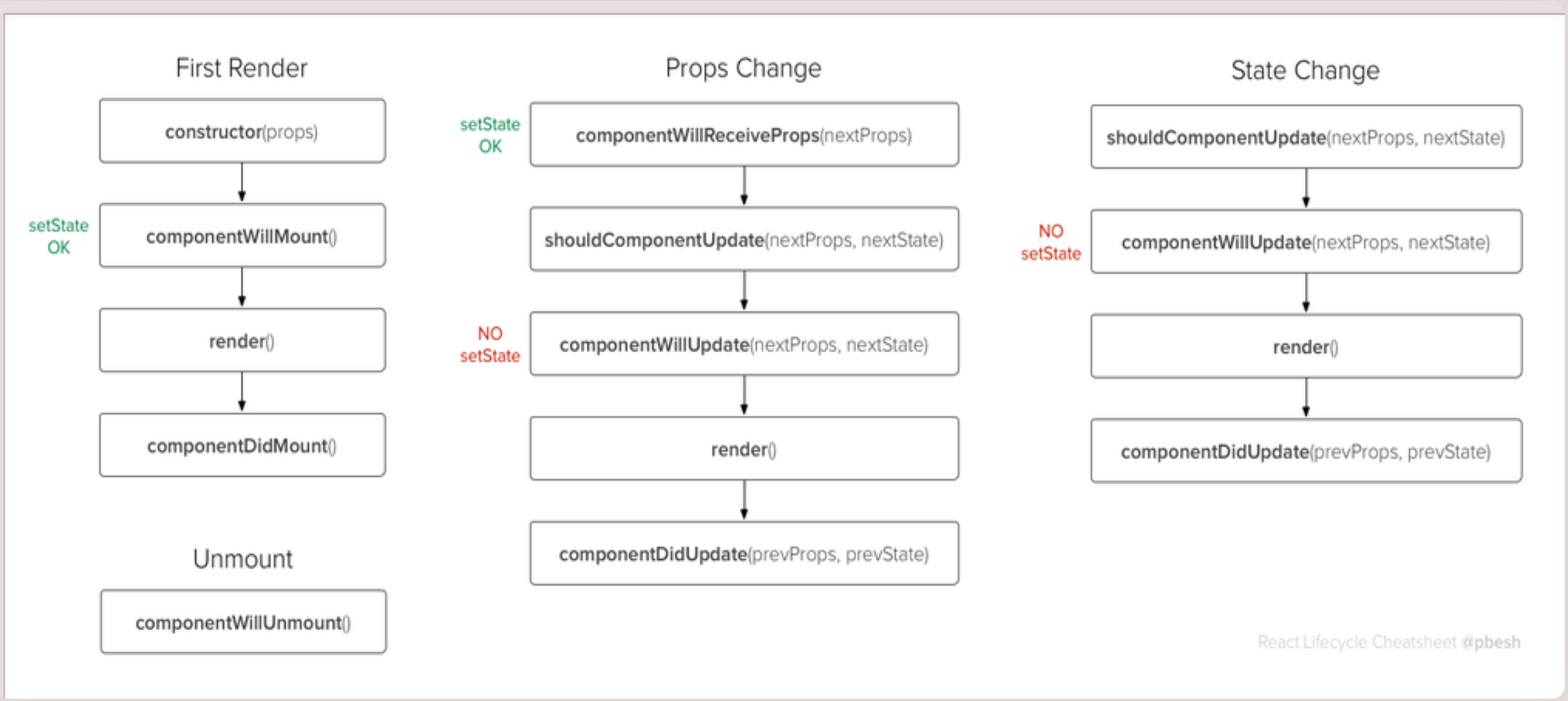
`Component` har mycket mer än bara `render()`

Det MESTA kommer du inte behöva använda. Men det finns där.

Ett antal funktioner kallas på automatiskt

Behöver vi inte funktionen, skriv inte ut!

Mest användbara är `componentDidMount()`


First Render
constructor(props)
setState OK
componentWillMount()
render()
componentDidMount()
Unmount
componentWillUnmount()
Props Change
setState OK
componentWillReceiveProps(nextProps)
shouldComponentUpdate(nextProps, nextState)
NO setState
componentWillUpdate(nextProps, nextState)
render()
componentDidUpdate(prevProps, prevState)
State Change
shouldComponentUpdate(nextProps, nextState)
NO setState
componentWillUpdate(nextProps, nextState)
render()
componentDidUpdate(prevProps, prevState)
React Lifecycle Cheatsheet @pbesh

Det vi behöver:

```
componentDidMount(){}
```

En såkallad **Lifecycle method**

En komponent monteras, renderas, uppdateras och när den inte ska finnas längre: avmonteras.

Den har en livscykel, den lever och dör som allt annat

Det finns speciella funktioner som körs på olika delar av livsspannet

```
componentDidMount(){
  fetch("https://example.com")
  .then(response => response.json())
  .then(data => {
    this.setState({ items: data });
  });
}
```

JSON måste plockas ut ur vårt response som är ett promise.

```
class App extends Component {
  state = {

  }
  componentDidMount(){
  }
  render(){

  }
}
```

```
class App extends Component {
  state = { }
  componentDidMount(){ }
  myOwnSuperMethod = () => { }
  render(){ }
}
```

Observera pilarna på den egna funktionen men inte på de inbyggda

Om vi hämtar informationen i `componentDidMount()` sker koden automatiskt och vi vet att komponenten finns

Så kalla inte på föregående kod i konstruktorn, det kan gå fel och är onödigt p.ga. orsaker. 🤖