express

Table of Contents

https://github.com/yduman/js-presentation

Package Managers 📦

Package Manager

- use `npm` or `yarn`
- manage dependencies, scripts, meta-informations, ...
- `package.json` is the first thing to look at

Sample package.json

```json
{
  "name": "js-presentation",
  "version": "1.0.0",
  "repository": "git@github.com:yduman/js-presentation.git",
  "author": "Yadullah Duman <yadullah.duman@gmail.com>",
  "scripts": {
    "start": "mdx-deck deck.mdx"
  },
  "devDependencies": {
    "mdx-deck": "^1.7.7"
  },
  "dependencies": { ... },
  "proxy": "..."
}
```

Documentation

How The Proxy Works

1. package.json has:

```
"proxy": "http://your-server"
```

2. React App calls `http://localhost:3000/api/users`

3. Dev server forwards call to `http://your-server/api/users`

Common Commands

Command	Description
`yarn init`	create a package.json file
`yarn add <lib>`	add a dependency
`yarn add --dev <lib>`	add a dev-dependency
`yarn <script_name>`	run a script
`yarn`	install dependencies

- ⚠ add `node_modules` to `.gitignore`

Lockfiles

- `yarn.lock` or `package-lock.json`
- ensure consistent installs across machines
- lockfiles contain the exact version of each dependency

- ⚠ always commit to your repository!

Express Backend 🖥

Basic Setup

- `yarn init`
- `yarn add express`
- `yarn add --dev nodemon`
- `yarn start`

```
"scripts": {
  "start": "nodemon ./src/server.js"
}
```

Hello World

```javascript
const express = require("express");
const http = require("http");

// init express app and supply it to an http server
const app = express();
const server = http.createServer(app);

app.get("/", (req, res) => {
  res.send({ hello: "world" });
});

const port = 8081;
server.listen(port, () => console.log(`Listening on port ${port}`));
```

Sample Project Structure

```
backend/
|__ node_modules/
|__ public/            Files you serve
|__ src/
|    |__ config/       Environment config
|    |__ db/           Database config
|    |__ middleware/   Own middlewares
|    |__ models/       DB Models
|    |__ routes/       REST routes
|    |__ tests/        Unit Tests
|    |__ utils/        Utility Functions
|    |__ server.js     Server Root
|__ package.json
|__ processes.json    PM2 config
|__ yarn.lock
```

- How to structure files?

HTTP-Verbs

Verb	Description
`GET`	should only retrieve data
`POST`	used to submit an entity to the specified resource
`PATCH`	used to apply partial modifications to a resource
`PUT`	replaces all current representations of the target resource with the request payload
`DELETE`	deletes the specified resource

Let's create an API!

- Backend for managing users
- We'll use some sort of authentication
- We'll use MongoDB along with Mongoose ODM

- MongoDB Docs
- Mongoose

Our API

HTTP	Endpoint	Description
`GET`	/api/user/me	get logged in user
`GET`	/api/user/:username	get user by username
`POST`	/api/user/	user registration
`POST`	/api/user/login	user login
`DELETE`	/api/user/:id	delete user by ID
`DELETE`	/api/user/logout	user logout

Setup

- install MongoDB and Robo3T
- install a REST Client

```
$ mkdir backend && cd backend
$ yarn init
$ yarn add express body-parser mongodb mongoose lodash jsonwebtoken bcryptjs
$ yarn add --dev nodemon chai mocha supertest
$ mkdir src && cd src && touch server.js
// add start script...
// edit server.js ...
$ yarn start
```

Start MongoDB and Robo3T

- Create a script that will start your DB server

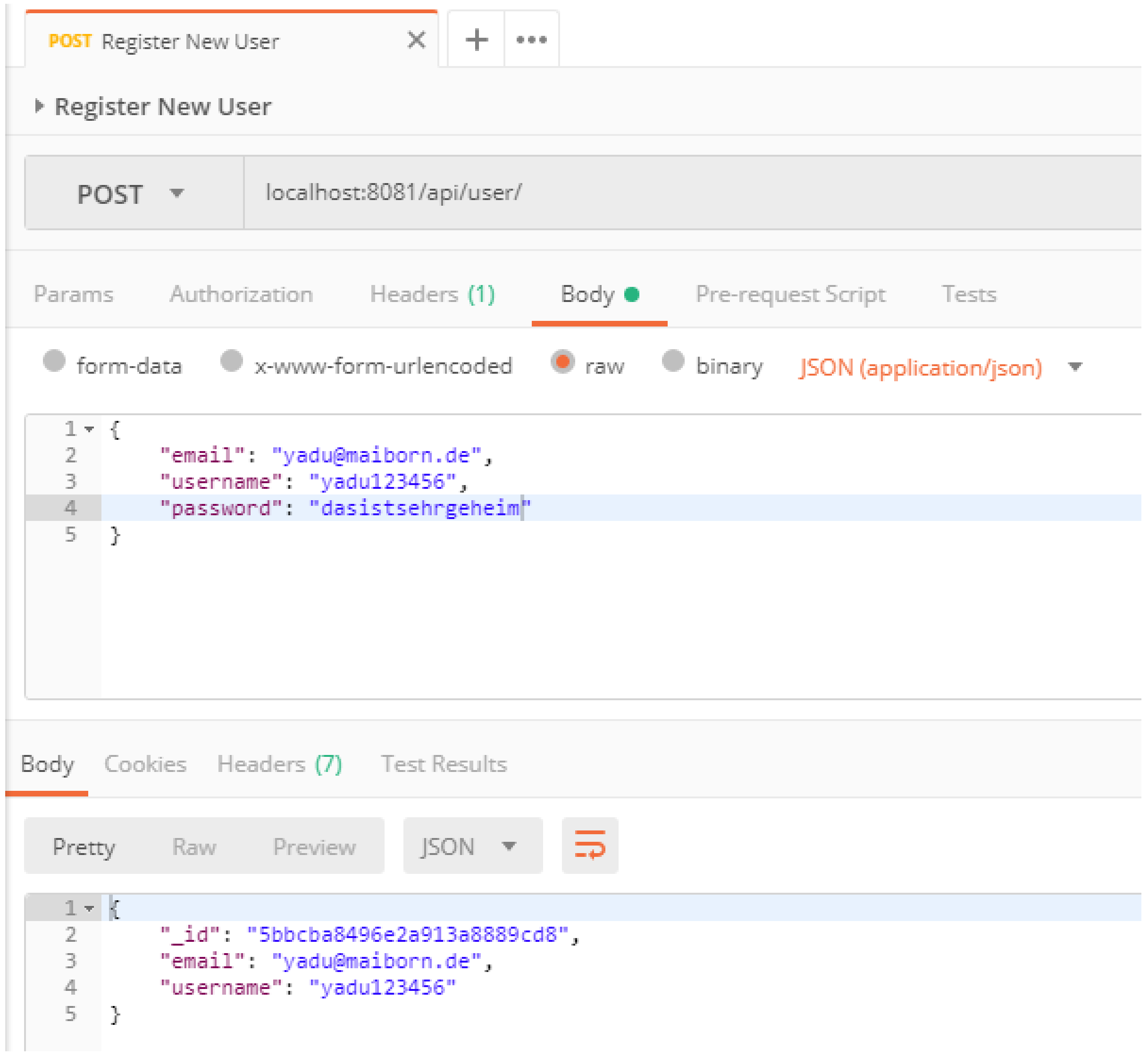
```
#!/bin/bash
mongod --dbpath /c/Users/yduman/mongo-data
```

```
$ sh start-mongo.sh
```

- Start Robo3T and connect to your local DB server

REST Clients

- e.g. Postman, Insomnia, ...



POST localhost:8081/api/user/

○ form-data ○ x-www-form-urlencoded ● raw ○ binary JSON (application/json)

```
{
    "email": "yadu@maiborn.de",
    "username": "yadu123456",
    "password": "dasistsehrgeheim"
}
```

JSON

```
{
    "_id": "5bbcba8496e2a913a8889cd8",
    "email": "yadu@maiborn.de",
    "username": "yadu123456"
}
```

Live Example 📺

Run Forever with PM2

```
{
  "apps": [{
    "name": "UserManagement",
    "script": "src/server.js",
    "watch": "src/",
    "ignore_watch": "src/uploads",
    "log_date_format": "YYYY-MM-DD HH:mm Z"
  }]
}
```

- PM2

React Frontend ⚛

Create React App

- *"Create React apps with no build configuration."*
- Version 2.0 is released since October 1st ♡

```
npx create-react-app <app_name>
```

Sample Project Structure

```
frontend/
|__ node_modules/
|__ package.json
|__ public/
|__ |__ favicon.ico
|__ |__ index.html
|__ |__ manifest.json
|__ src/
|   |__ components/   All components
|   |__ styles/       All stylings
|   |__ utils/        All utility functions
|   |__ index.js
|   |__ registerServiceWorker.js
|__ yarn.lock
```

Main Concepts ☝

Components and Props

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

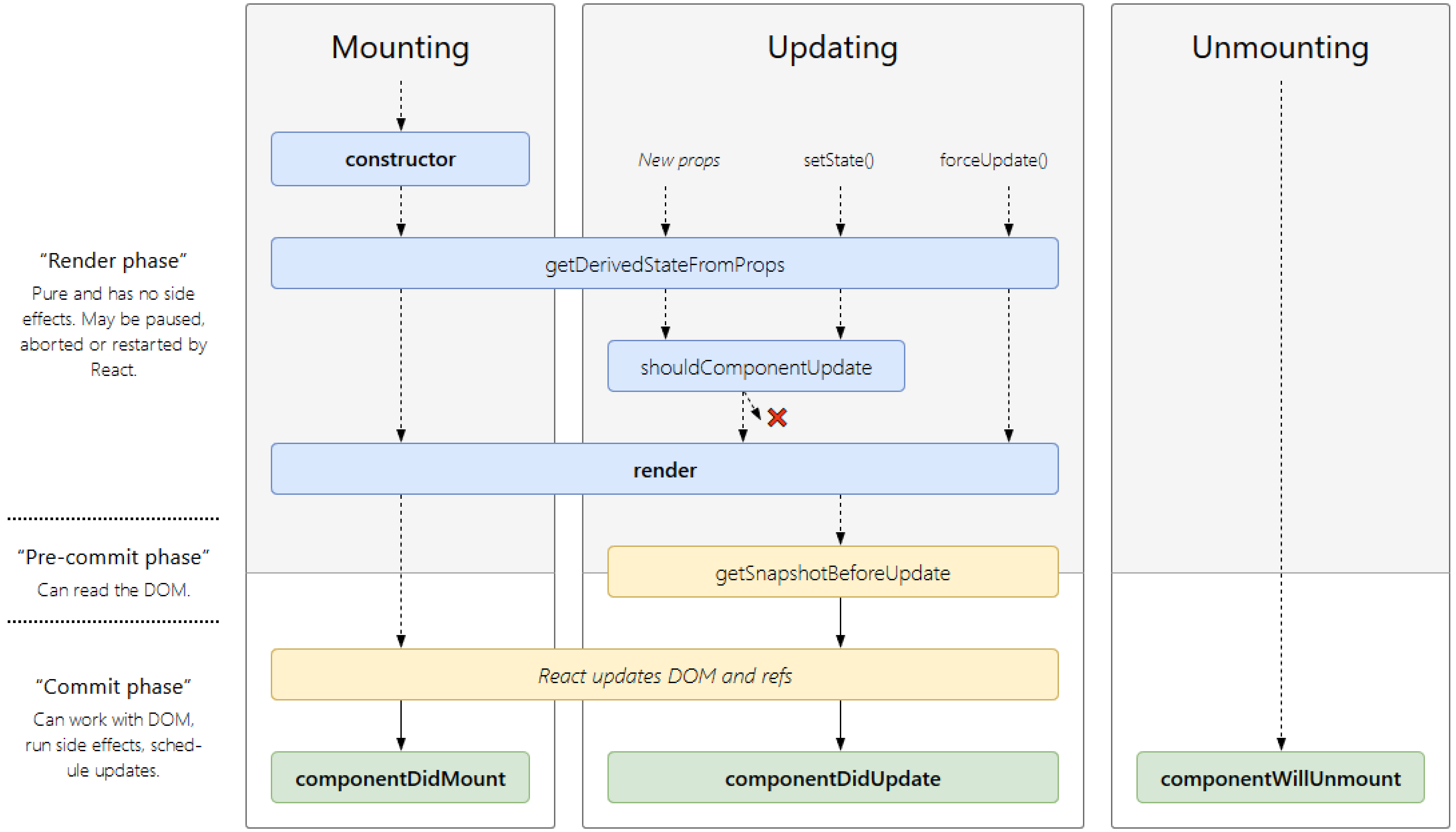
Functional and Class Components

```
// Functional Component
const Welcome = props => <h1>Hello, {props.name}</h1>;
```

```
// Class Component
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

```
const App = () => <div><Welcome name="John" /></div>;
ReactDOM.render(<App/>, document.getElementById("root"));
```

State and Lifecycle



"Render phase"
Pure and has no side effects. May be paused, aborted or restarted by React.

"Pre-commit phase"
Can read the DOM.

"Commit phase"
Can work with DOM, run side effects, schedule updates.

State

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

  render() {
    return (
      <div>
        <h1>Hello World!</h1>
        <h2>It is { this.state.date.toLocaleTimeString() }</h2>
      </div>
    );
  }
}
```

State and Lifecycle

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

  componentDidMount() {
    this.timerID = setInterval(() => this.setState({ date: new Date() }), 1000);
  }

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

  render() {
    return (
      <div>
        <h2>It is { this.state.date.toLocaleTimeString() }</h2>
      </div>
    );
  }
}
```

Handling Events

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

```
class Incrementer extends React.Component {
  state = { counter: 0 };

  handleClick = event => {
    event.preventDefault();

    this.setState(prevState => {
      return { counter: prevState.counter + 1 };
    });
  };

  render() {
    const { counter } = this.state;
    return (
      <div>
        <p>{ counter }</p>
        <button onClick={this.handleClick}>Increment</button>
      </div>
    );
  }
}
```

Conditional Rendering

In React, you can create distinct components that encapsulate behavior you need.
Then, you can render only some of them, depending on the state of your application.

Conditional Rendering

```
const UserGreeting = () => <h1>Welcome Back!</h1>;

const GuestGreeting = () => <h1>Please sign up.</h1>;

const Greeting = props => {
  const isLoggedIn = props.isLoggedIn;

  if (isLoggedIn)
    return <UserGreeting />;

  return <GuestGreeting />;
}

ReactDOM.render(<Greeting isLoggedIn={false} />, document.getElementById("root"));
```

Lists and Keys

```
const NumberList = props => {
  const numbers = props.numbers;
  const listItems = numbers.map(num => <li key={num.toString()}>{ num }</li>);

  return <ul>{ listItems }</ul>
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(<NumberList numbers={numbers} />, document.getElementById("root"));
```

- Keys help React identify which items have changed, are added, or are removed.
- Keys should be given to the elements inside the array to give the elements a stable identity.

Forms

```
class NameForm extends React.Component {
  state = { name: "" };
  handleChange = event => this.setState({ name: event.target.value });
  handleSubmit = event => {
    event.preventDefault();
    alert("Submitted Name: " + this.state.name);
  };
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.name} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Thinking in React

1. Break the UI into a Component Hierarchy
2. Build a static version in React
3. Identify the minimal representation of UI state
4. Identify where your state should live
5. Add inverse data flow

Source

Build a static version in React

- Build a version that takes your data model and renders the UI but has no interactivity
- Don't use state at all to build this static version
- State is reserved only for interactivity
- Pass data using `props`

Identify the minimal representation of UI state

- Don't Repeat Yourself
- Think of the minimal set of mutable state that your app needs
- If you're for example building a TODO list, just keep an array of the TODO items around and don't keep a separate state variable for the count, just compute the length of the array

Our Data

1. The original list of products
2. The search text the user has entered
3. The value of the checkbox
4. The filtered list of products

Figure out State

1. Is it passed in from a parent via props? If so, it probably isn’t state.
2. Does it remain unchanged over time? If so, it probably isn’t state.
3. Can you compute it based on any other state or props in your component? If so, it isn’t state.

What remains

2. The search text the user has entered
3. The value of the checkbox

Identify where your state should live

- Identify every component that renders something based on that state
- Find a common owner component
- Either the common owner or another component higher up in the hierarchy should own the state

Add inverse data flow

- Now it's time to support data flowing the other way
- `FilterableProductTable` will pass callbacks to `SearchBar` that will call `setState()`

Testing in React 💉

react-testing-library by Kent C. Dodds

- encourages better testing practices
- tests will work with the actual DOM, rather than rendered React components
- query the DOM in the same way the user would
- exposes a recommended way to find elements by a `data-testid`

- GitHub
- TDD with react-testing-library

E2E-Testing with Cypress

```
describe('My First Test', () => {
  it('Gets, types and asserts', () => {
    cy.visit('https://example.cypress.io')
    cy.contains('type').click()
    cy.url().should('include', '/commands/actions')
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com')
  })
})
```

- Website
- cypress-testing-library by Kent C. Dodds

Deployment

Deployment

- build your React App with `yarn build`
- move built files to the `public` folder of your Express App
- deploy it somewhere and manage it with PM2

```
// server.js
// ...
app.use(express.static(__dirname + "/public"))
// ...
```

Live Example 📺

Some Neat Libraries 📚

- react-testing-library
- react-beautiful-dnd
- react-responsive-modal
- react-fontawesome
- react-pose
- prop-types
- @reach/router
- mdx-deck
- styled-components
- bulma
- material-ui
- and much more 😉

Some Advice 🧙

- 👩‍💻 Learning by doing!
- 🤓 Be curious, try out new stuff!
- 🔍 Google is your friend
- 📝 Use your notepad, if stuck
- 📰 Read Blogs
- 💻 Contribute to Open Source
- 🎓 Participate on Online Courses

Happy Hacking!