Part 1

Make a React app with create-react-app

```
npx create-react-app foldername
```

You can run with

```
npm start
```

Node

Node.js is a Javascript runtime environment. You can run it with

```
node name_of_file.js
```

Or type node into the command line, or a browser developer tool console.

Destructuring props

```
props = {
  name: 'Arto Hellas',
  age: 35,
}

const { name, age } = props

const Hello = ({ name, age }) => {
```

Object spread syntax for copying an object and changing part of it

```
const clicks = {
  left: 0, right: 0
}
```

Copies clicks, and adds 1 to left

```
const newClicks = {
  ...clicks,
  left: clicks.left + 1
}
```

Function that return a function

Useful if you want to make multiple event handlers that are similar to each other, but not exactly alike

https://fullstackopen.com/en/part1/a_more_complex_state_debugging_react_apps#function-that-returns-a-function

I don't care if the folder is full, delete it

For example, if you used create-react-app but you didn't want it to have its own git repository, run

```
rm -rf .git
```

Part 2

Autocomplete console.log
log + tab

Importing modules
`import defaultExport from "module-name";` <= Import entire module
`import { export1 } from "module-name";` <= Import a single export

Exporting modules/stuff from modules

```
export default Note
export { myFunction, myVariable };
export const myFunction = () => …
```

Forms, controlled forms
Add an onSubmit event handler to the form itself, and an onChange to the input fields. Save the values in state. Set the values of input fields to state (thus controlled form). On submit, prevent event default and use field values in state to make your object, whatever.

Filtering results

```
const notesToShow = showAll ? notes : notes.filter(note =>
note.important === true)
```

Add devDependency

```
npm install <package-name> --save-dev
```

Scripts in package.json
Stuff that was in there originally, like start: `npm start`
Stuff that wasn't (scripts you added) npm run nameofscript

Using JSON server
Make a fake database file in the root directory (not inside src, the main folder!) called db.json and put some stuff in it. For example, an object with an array inside it {notes: ["one", "two", "three]}
Install json-server as a dev dependency.
To keep it running,

```
npm run json-server --watch db.json --port 3001
```

package.json script==>

```
"server": "json-server --watch db.json --port 3001"
```

Your "notes" array will be at http://localhost:3001/notes
More details on how it behaves https://github.com/typicode/json-server

Axios
Install axios for making requests, and import it as needed

```
import axios from 'axios'
```

eventually separate requests out to a services folder. One file for each endpoint?

blogs.js for doing stuff to “/api/blogs,” etc

useEffect set to empty array, for when the component first mounts, and event handlers are good places to tell axios to do things.

It’s easiest to save a base url

```
const baseUrl = "/api/blogs";
```

You can write as an async function or not, just return response data.

```
const getAll = () => {
  const request = axios.get(baseUrl);
  return request.then((response) => response.data);
};

const deleteBlog = async (id) => {
  const response = await axios.delete(`${baseUrl}/${id}`,
config);
  return response.data;
};
```

Shorthand, config optional

```
axios.get(url[, config])
axios.delete(url[, config])
axios.post(url[, data[, config]])
axios.put(url[, data[, config]])
axios.patch(url[, data[, config]])
```

Longhand, for reference

```
axios({
  method: 'post',
  url: '/login',
  data: {
    firstName: 'Finn',
    lastName: 'Williams'
  }
});
```

The response will contain the new added whatever it is in response.data.

Adding an error handler to a promise chain

blogHandler.getAll().then(blogs=> stuff).catch(error=> console.log(error));

Replacing a changed member of an array

```
updatedArray = notes.map(note => note.id === newNoteId ?
newNote : note)
```

Saving an api key in process.env

Make an .env file and put your api key in it

```
REACT_APP_API_KEY='t0p53cr3t4p1k3yv4lu3'
```

You can get the key in your code with `process.env.REACT_APP_API_KEY`

Make sure you have a .gitignore file with .env (and node_modules for that matter)

Timed error message

Make a message component that receives a message as props, returns null if message is null and the component if otherwise

Store message in state in a parent component. Then have a function for displaying message, and a timeout for resetting the message to null.

```
 setMessage(`This is my message`)
setTimeout(() => {setErrorMessage(null)}, 5000)
```

Put message component in appropriate spot inside render so it will pop up as needed.

Part 3

Implementing the backend! Node!

https://fullstackopen.com/en/part3/node_js_and_express

Go to the appropriate directory, type `npm init`, and answer ?s

Make sure main is index.js. not much else is important

Open resulting package.json and add `"start": "node index.js"` to scripts.

Make an index.js file

Install express (`npm install express`)

Importing in Node

const whathaveyou = require("thing");

Express is the less cumbersome way to make a web server in Node than the built-in http kind. It can guess the right status code and the content type so you don't have to specify, and it will change things into JSON format for you if you call response.json

https://fullstackopen.com/en/part3/node_js_and_express#web-and-express

Basic app is

```
const express = require('express')
const app = express()
```

App needs to use express.json() middleware to parse JSON data coming from incoming requests.

"The json-parser functions so that it takes the JSON data of a request, transforms it into a

JavaScript object and then attaches it to the *body* property of the *request* object before the route handler is called."
So incoming JSON data gets turned into a {name: "Satsuki", color: "black"} object living inside request.body. So you can get name from request.body.name, for example.

```
app.use(express.json())
```

Controllers
Then, various responses to .get and .post and whatever,

```
app.get('/', (request, response, next) => {
  response.status(201).json(populatedBlog);
})
```

Usually it'll be response.json(some data), unless you need to specify something other than 200 OK.
Express sends 200 OK as a default, and can automatically send some error codes too
If you want to just send a # response, do response.status(200).end()
End is "Used to quickly end the response without any data"

Put in error handling:

```
    .catch(error => next(error))
```

and add next to (request, response, next) if you haven't yet.
Of course you'll need an error handler to do anything with that next

Getting a parameter out of a route

```
app.get('/api/notes/:id', (request, response) => {
  const id = request.params.id
….
```

Statuses
200 OK
201 Created: good response for after POST/PUT request
204 No content: a nice response for successful DELETE request
400 Bad request: for incomplete/wrong request. Like you need a {title: "Blah" author: "Blah"} but there's no author or something. Too short password. Etc.
401 Unauthorized: we needed a token but the token's messed up
404 Not found: unknown endpoint
418 I'm a teapot: I don't make coffee

Listening to port
then define the port and ask app to listen to that port.

```
const PORT = 3001
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})
```

Eventually these will be split out into controllers for .get .post etc, listening at PORT

will be in index.js

Set up **nodemon** as a development dependency so you don't have to constantly restart the application to see changes
`npm install --save-dev nodemon`

Add `"dev": "nodemon index.js"` to scripts so you can start your app with nonodemon with
`npm run dev`

Testing one: using Postman and the Visual Studio Code REST client
These are both used to make test requests, and we prefer VS Code. Put requests in a requests folder, inside a rest file. all_requests.rest

Separate with a ###, then the name of method, then URL, content type, a new line, the data you're sending.

```
###
POST http://localhost:3003/api/blogs
Content-Type: application/json

{
    "title": "Budget Bytes",
    "author": "Beth",
    "url": "https://www.budgetbytes.com/",
    "likes": 5
}
```

MIddleware
Put middleware inside a utils folder.
Other middleware besides express.json needs to go after it somewhere, otherwise we won't get content of request parsed out of JSON.
You use middleware by importing it, then calling app.use(middlewareName);
Some middleware functions will go before the routes are called: the logger, and the token extractor if we have one.
Some middleware will come after routes: the unknown endpoint handler, and the error handler. Error handler has to be last.

Connecting front end to back end locally, and being thwarted by CORS
Install cors
`npm install cors`

```
const cors = require('cors')
app.use(cors())
```

Cors middleware goes before express.json

Deploying to Heroku

Add a file called *Procfile* to the project's root to tell Heroku how to start the application.

```
web: npm start
```

Change port to `process.env.PORT || 3001` so it's either default 3001 locally, or whatever process.env.PORT set inside Heroku.

Make a .gitignore file in the root folder that ignores node_modules

Create a Heroku application with the command `heroku create`, commit your code to the repository and move it to Heroku with command *git push heroku main*.

You can access continual heroku logs with `heroku logs -t`

If you want to host front and backend in the same spot

See here (part 3b, serving static files from the backend):
https://fullstackopen.com/en/part3/deploying_app_to_internet#serving-static-files-from-the-backend

Change urls to backend to relative urls, run a build, copy build folder to backend, and use express static middleware

```
app.use(express.static('build'))
```

There are a bunch of scripts to simplify updating the frontend on Heroku, and how to still connect to local server in dev move despite relative URLs. See link.

MongoDB

https://fullstackopen.com/en/part3/saving_data_to_mongo_db#backend-connected-to-a-database

Create a cluster. Choose AWS and one of the free regions.

When it's ready, go into database access and set a username and password with privileges. Whitelist all IP addresses (not that it'll do you much good)

Click connect. MongoDB will give you some bullshit connection string that won't work. Use the slightly older one.

Mongoose

We will use Mongoose to talk with MongoDB. Install mongoose.

inside App

```
const mongoose = require("mongoose");
const config = require("./utils/config");

const mongoUrl = config.MONGODB_URI;

mongoose
  .connect(mongoUrl, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    useFindAndModify: false,
    useCreateIndex: true,
  })
  .then(() => {
```

```
    logger.info("Successfully connected to the database");
  })
  .catch((err) => {
    logger.info("Didn't connect to database and here's the
error", err);
  });
```

The connect URL and PORT lives insides a .env file, accessed by config inside the utils folder. Config needs dotenv to access the .env file
Gitinore the .env file or else!!1

Inside config

```
require("dotenv").config();

let MONGODB_URI = process.env.MONGODB_URI;
if (process.env.NODE_ENV === "test") {
  MONGODB_URI = process.env.MONGODB_URI_TEST;
}
const PORT = 3001;

module.exports = { MONGODB_URI, PORT };
```

Making schemas
We need to define the shape of the documents within each collection (a Schema), and then make a constructor for making new ones (a model)
Make a models folder and define each in a lower case, singular file
blog.js
Name of schema: blogSchema
Name of model: Blog
Name of collection up on MondoDB (automatic): blogs

A simple schema

```
const mongoose = require("mongoose");

const blogSchema = new mongoose.Schema({
  title: String,
  likes: Number
});

blogSchema.set("toJSON", {
  transform: (document, returnedObject) => {
```

```
    returnedObject.id = returnedObject._id.toString();
    delete returnedObject._id;
    delete returnedObject.__v;
  },
});

module.exports = mongoose.model("Blog", blogSchema);
```

Setting toJSON means we can edit what gets passed back. We can delete private _id and _v and change _id from its original object to a string.

Working with MongoDB inside controllers
Import the models you need and use commands like findOne, findById, .save(), use the constructor as needed to make new mongo objects. These commands are async so you'll need to for example, make that controller async and use await.

```
const user = await User.findById(decodedToken.id);
```

findByIdAndUpdate usually returns the original document, so set it to return changed document if that's what you need.

```
Note.findByIdAndUpdate(request.params.id, note, { new: true })
```

The id, the new document, the new setting.

Setting environment variables in Heroku
Either use the command line

```
heroku config:set MONGODB_URI=mongodb+srv://
fullstack:secretpasswordhere@cluster0-ostce.mongodb.net/note-
app?retryWrites=true
```

or set it inside Heroku dashboard in browser. Settings => config variables.

Part 4

Testing Node applications!

```
npm install --save-dev jest
```

add script:

```
  "test": "jest --verbose"
```

Then either add

```
"jest": {
   "testEnvironment": "node"
 }
```

to the end of package.json, or make a new file *jest.config.js* containing this:

```
module.exports = {
  testEnvironment: 'node',
```

```
};
```

Make a tests directory and name your tests testname.test.js

The test:
import your function
`const palindrome = require('../utils/for_testing').palindrome`
The thing inside parans is relative path to file, for_testing is file name, .palindrome is name of the function

two tests inside a describe block, with import

```
const average = require('../utils/for_testing').average

describe('average', () => {
  test('of one value is the value itself', () => {
    expect(average([1])).toBe(1)
  })

  test('of many is calculated right', () => {
    expect(average([1, 2, 3, 4, 5, 6])).toBe(3.5)
  })

})
```

Run npm test (script added above) to do all, or run a single test with only method (describe.only or test.only)
Or, use this to run a single test or block `npm test -- -t 'when list has only one blog, equals the likes of that'`
You can just type part of the name and that'll run whatever contains that fragment

Testing the backend app with a fake API ("integration tests")
We need to change so that app runs on either production or development mode, with a corresponding change to which database we access in MongoDB

Add scripts
```
"start": "NODE_ENV=production node index.js",
"dev": "NODE_ENV=development nodemon index.js",
```
add/replace script:
```
"test": "NODE_ENV=test jest --verbose --runInBand"
```

Add to config file

```
const MONGODB_URI = process.env.NODE_ENV === 'test' ?
process.env.TEST_MONGODB_URI  : process.env.MONGODB_URI
```

Add URL to test mongo db in .env file (remember new databases just appear when you change the name part inside URL)

Install supertest

```
npm install --save-dev supertest
```

Imports and stuff

```
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')
const Blog = require("../models/blog")

const api = supertest(app)
```

A sample test with before each and close connection at the end. testHelper.blogs is a list of objects that was imported. The Blog model (the constructor) also need to be imported.

```
beforeEach(async () => {
  await Blog.deleteMany({});
  const blogObjects = testHelper.blogs.map((blog) => new
Blog(blog));
  const promises = blogObjects.map((blogObject) =>
blogObject.save());
  await Promise.all(promises);
});

test("Successfully gets correct number of blogs", async () =>
{
  const response = await api.get("/api/blogs").expect(200);
  expect(response.body).toHaveLength(testHelper.blogs.length);
});

afterAll(() => {
  mongoose.connection.close();
});
```

Using async/await for controller without try/catch

```
npm install express-async-errors
```

Put library in app.js

```
require('express-async-errors')
```
If you do that, you can get rid of next and the try-catch blocks in each router (controllers)
"The library handles everything under the hood. If an exception occurs in a *async* route, the execution is automatically passed to the error handling middleware."

References across mongoDB collections

```
const userSchema = new mongoose.Schema({
  username: String,
  name: String,
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ],
})
```

Add id for notes as needed. When you return, populate the notes (assign 1 to whichever fields you want)
Remember, inside MongoDB the id is _id, you call it id and make it a string when you *return* it.

```
const users = await User.find({}).populate("notes", { content: 1, date: 1 });
```

User administration

Never store unencrypted plain text passwords in a database.

```
npm install bcrypt
```

Making a new user: the router's post method will get the password from the body object, encrypt it, and save that.

```
const body = request.body

const saltRounds = 10
const passwordHash = await bcrypt.hash(body.password, saltRounds)

const user = new User({
  username: body.username,
  name: body.name,
  passwordHash,
})
```

Validation with MongoDB
Some stuff can be checked with built in validators, but uniqueness (like unique usernames) requires an outside library

```
npm install mongoose-unique-validator
```

```
const uniqueValidator = require('mongoose-unique-validator')
…
const userSchema = new mongoose.Schema({
  username: {
    type: String,
    unique: true  },
  name: String,
  passwordHash: String
}

userSchema.plugin(uniqueValidator)
```

required, min/max for numbers, minLength and maxLength for strings etc are built in. unique is not and comes from mongoose-unique-validator

Token authentication: login

```
npm install jsonwebtoken
```

Inside login controller

```
const jwt = require('jsonwebtoken')
const bcrypt = require('bcrypt')

….
const user = await User.findOne({ username: body.username })
  const passwordCorrect = user === null ? false : await bcrypt.compare(body.password, user.passwordHash)

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  const userForToken = {
    username: user.username,
```

```
    id: user._id,
  }

  const token = jwt.sign(userForToken, process.env.SECRET)
```

!!!....breaking news....!!!
The new 2021 version adds a section where the token expires in an hour, also adding another block to errorhandler to handle an expired token

```
const token = jwt.sign(
  userForToken,
  process.env.SECRET,
  { expiresIn: 60*60 }
  )
```

!!!....breaking news....!!!

```
  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
```

Set the value for SECRET in your process.env file. If the username exists and the password fits passwordHash, jsonwebtoken returns a signed token with process.env.SECRET and whatever info the front end needs to save about user.

Requiring a signed token from the user's end
We used Authorization header where the Authorization header has value
`Bearer <~~~the long ass token~~~>`

Use a helper function inside the router to check token

```
const jwt = require("jsonwebtoken");

const getTokenFrom = (request) => {
  const authorization = request.get("authorization");
  if (authorization &&
authorization.toLowerCase().startsWith("bearer ")) {
    return authorization.substring(7);
  }
  return null;
};
```

inside whatever requires a token…

```
  const decodedToken = jwt.verify(token, process.env.SECRET);
  if (!token || !decodedToken.id) {
    return response.status(401).json({ error: "token missing
or invalid" });
  }
  const user = await User.findById(decodedToken.id);
```

Note about security
The app uses an http server, but that's OK: "Heroku routes all traffic between a browser and the Heroku server over HTTPS."

Part 5

Destructuring target from event handler in form fields

```
onChange={({ target }) => setPassword(target.value)}
```

Setting/getting a token etc from local storage
In our example, this is user object sent back from backend with username, name, and token
Set

```
window.localStorage.setItem("loggedInUser",
JSON.stringify(user));
```

Get

```
const savedUser =
JSON.parse(window.localStorage.getItem("loggedInUser"));
```

A common pattern is then setting state to savedUser || null

Saving and using token in services
as a global variable inside ex. noteService

```
let token = null
const setToken = newToken => {
     token = `bearer ${newToken}`
}
```

a function that needs token. Now post has URL, new object, and authorization with the token

```
const create = async newObject => {
  const config = { headers: { Authorization: token },  }
  const response = await axios.post(baseUrl, newObject,
config)
  return response.data
}
```

setToken is called from app when loaded to check for user from localStorage, and upon successful login (inside login handler). Local storage also called inside login handler to save

A component that wraps its children (props.children)
Use props.children in the render method and your component can contain other components
<Ocean name="Pacific">
<Fish />
</Ocean>

Accessing a child component variable from parent
Without pulling it further up? Can do with useRef, See https://fullstackopen.com/en/part5/props_children_and_proptypes#references-to-components-with-ref

Testing React apps ("unit tests")
`npm install --save-dev @testing-library/react @testing-library/jest-dom`

simple example. The render method we used renders the components in a format that is suitable for tests without rendering them to the DOM.
render returns an object that has several properties. One of the properties is called *container*, and it contains all of the HTML rendered by the component.

```
import React from 'react'
import '@testing-library/jest-dom/extend-expect'
import { render } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const component = render(
    <Note note={note} />
  )
```

```
  expect(component.container).toHaveTextContent(
    'Component testing is done with react-testing-library'
  )
})
```

npm test will start the tests, which will then hang there, watching, until you make a change.
Tests get stored in the same folder as component because that's how it's configured by default, but might consider changing that because I kind of hate it.

Firing events and mock functions

```
import { render, fireEvent } from '@testing-library/react'

// ...

test('clicking the button calls event handler once', () => {
  const note = {
    content: 'Component testing is done with react-testing-
library',
    important: true
  }

  const mockHandler = jest.fn()

  const component = render(
    <Note note={note} toggleImportance={mockHandler} />
  )

  const button = component.getByText('make not important')
  fireEvent.click(button)

  expect(mockHandler.mock.calls).toHaveLength(1)
})
```

Finding elements based on text (getByText) is the most foolproof way. You can also add class names or ids to make it easier to grab things, or use querySelector to get a type of element .querySelector("form")

Check your test coverage with
`CI=true npm test -- --coverage`

End to end testing
Cypress!
in frontend:
`npm install --save-dev cypress`
new script:
`"cypress:open": "cypress open"`

new script in backend
`"start:test": "NODE_ENV=test node index.js"`

start up both (remember backend in test mode, so npm run start:test) and cypress will make some folders

Delete the tests in integration/examples and add your own tests name.spec.js

simple example

```
describe('Note app', function() {
  beforeEach(function() {    cy.visit('http://localhost:3000')  })

  it('front page can be opened', function() {
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2021')
  })

  it('login form can be opened', function() {
    cy.contains('log in').click()
  })

})
```

Note you use .contains to check if it's there, and also click it or whatever. Can also .get by id.

```
cy.contains('log in').click()
cy.get('#username').type('mluukkai')
```

End to end testing and the database
E2E tests don't have access to database so you need to add a

new router for the tests that is in the backend for test mode only.
Look here
https://fullstackopen.com/en/part5/end_to_end_testing#controlling-the-state-of-the-database

Cypress is too slow
Use .only to run one test at a time

Bypass the UI
Only login through the UI once and then use cypress to directly post to login or whatever

Custom commands
Separate out commonly used actions into *cypress/support/commands.js*.

Part 6

Redux, which we've done a lot lately, so very briefly:

```
npm install redux
npm install react-redux
npm install redux-thunk
npm install --save-dev redux-devtools-extension
```

store.js

```
import { createStore, combineReducers, applyMiddleware } from "redux";
import thunk from "redux-thunk";
import { composeWithDevTools } from "redux-devtools-extension";

import messageReducer from "./reducers/messageReducer";
import blogReducer from "./reducers/blogReducer";
import userReducer from "./reducers/userReducer";
import usersReducer from "./reducers/usersReducer";

const reducer = combineReducers({
  message: messageReducer,
  blogs: blogReducer,
  user: userReducer,
  users: usersReducer,
});

const store = createStore(reducer,
```

```
composeWithDevTools(applyMiddleware(thunk)));

export default store;
```

index.js

```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import store from "./store";
import App from "./App";
import "./index.css";

ReactDOM.render(
  <Provider store={store}>
        <App />
  </Provider>,
  document.getElementById("root")
);
```

Sample async action creator with thunk

```
export const setAllBlogs = () => {
  return async (dispatch) => {
    const blogs = await blogService.getAll();
    dispatch({
      type: "SET_ALL_BLOGS",
      data: { blogs },
    });
  };
};
```

Sample reducer

```
const blogReducer = (state = [], action) => {
  switch (action.type) {
    case "SET_ALL_BLOGS":
      return action.data.blogs;
    case "ADD_BLOG":
      return state.concat(action.data.addedBlog);
    case "UPDATE_BLOG":
      return state.map((item) =>
```

```
        item.id === action.data.updatedBlog.id ?
action.data.updatedBlog : item
      );
    default:
      return state;
  }
};

export default blogReducer;
```

Use action creator inside component

```
  const dispatch = useDispatch();

dispatch(setAllBlogs(updatedBlogs));
```

Getting store inside component

```
  const blogs = useSelector((state) => state.blogs);
```

(skip connect, older version of useDispatch, useSelector)

Part 7

React-router

```
npm install react-router-dom
```

inside index.js

```
import { BrowserRouter as Router } from "react-router-dom";

ReactDOM.render(
    <Router>
        <App />
    </Router>
  document.getElementById("root")
);


inside app (including routeMatch)
import { Switch, Route, useRouteMatch, Redirect } from "react-
router-dom";

 const match = useRouteMatch("/users/:id");
```

```
  const matchUser = match
    ? users.find((user) => user.id === match.params.id)
    : null;

    <Switch>
        <Route path="/users/:id">
          {matchUser ? <User user={matchUser} /> : <Redirect
to="/users" />}
        </Route>
        <Route path="/blogs">
          <Blogs showMessage={showMessage} />
        </Route>
    </Switch>

inside wherever your menu is
import { Link } from "react-router-dom";

<Link to="/">home</Link>
<Link to="/notes">notes</Link>
<Link to="/users">users</Link>
```

Custom hooks

example: useField for form fields

```
const useField = (type) => {
  const [value, setValue] = useState('')

  const onChange = (event) => {
    setValue(event.target.value)
  }

  return { type, value, onChange }
}

use elsewhere like regular hook

const name = useField('text')

can use spread attributes to plug everything into input
<input type={name.type} value={name.value}
onChange={name.onChange} />;
```

becomes

```
<input {...name} />
```

Styles
Overview of React Bootstrap, Material UI, and styled components. I preferred Material UI with custom color scheme (see part 7 extended-bloglist)

Webpack
See webpack-practice example for react, odin project notes for “vanilla” Javascript.

Miscellaneous grab bag
Class components, organization of code, where to put front end vs back end, polling and websockets, security, future trends, library and link round-up.