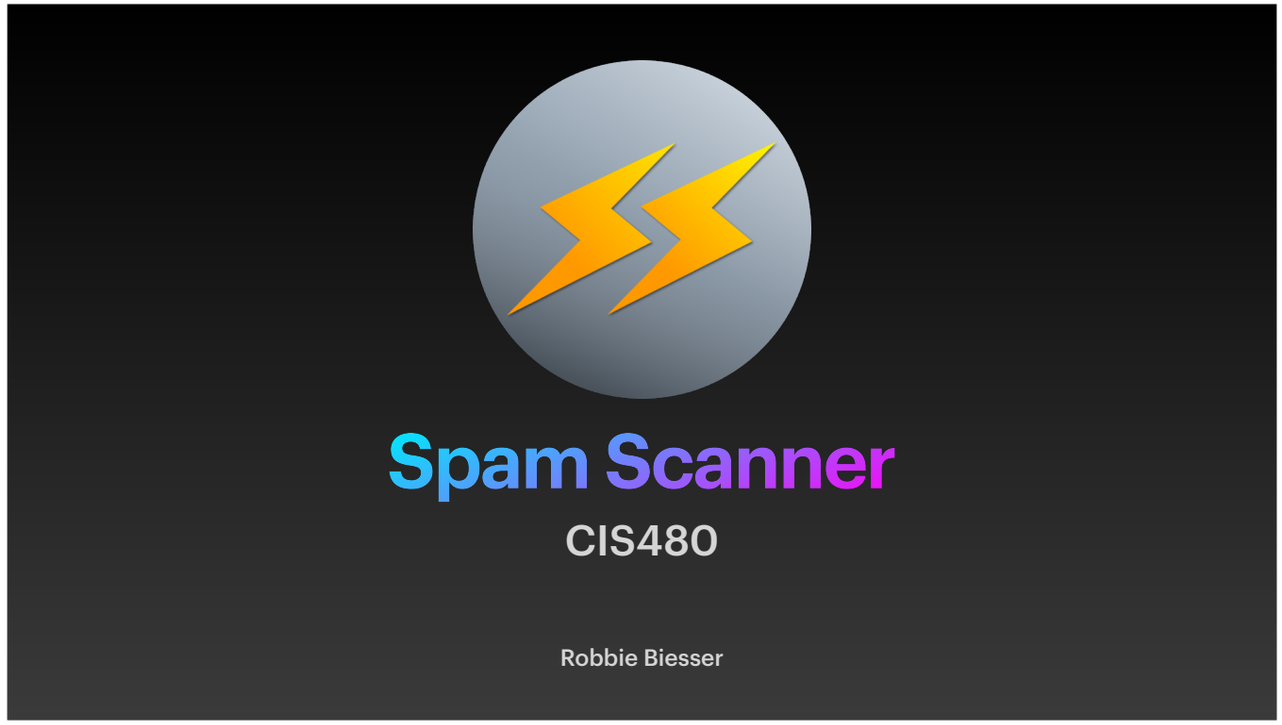
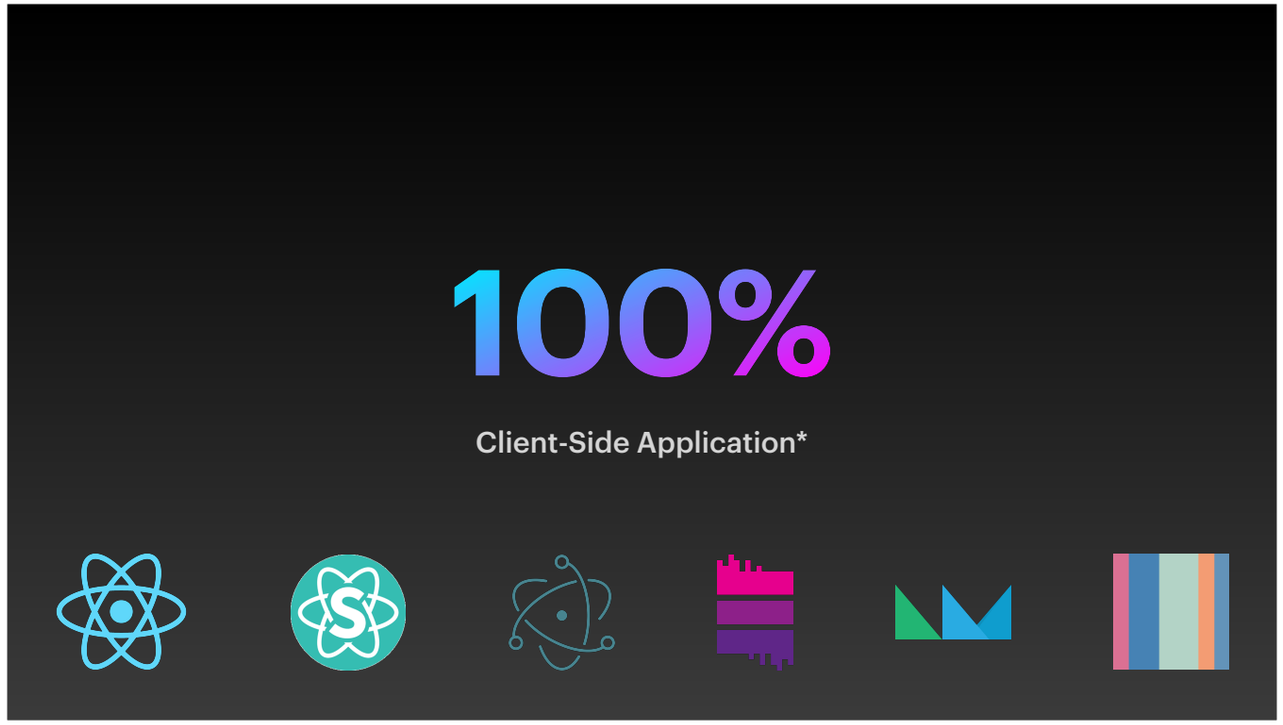
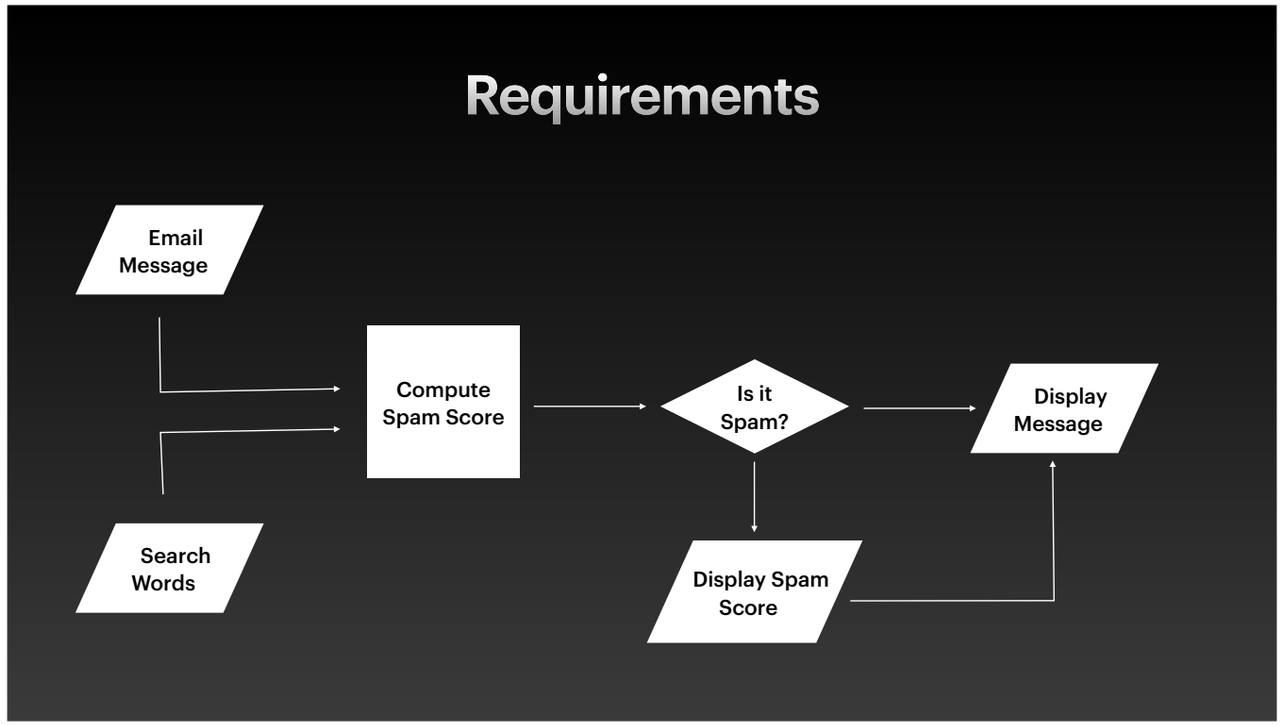


The idea for Spam Scanner spawned from an interesting problem statement, "Spam costs U.S. organizations billions of dollars a year in spam prevention systems in addition to the consumption of system resources and decreased productivity." Spam is a term given to unsolicited and unwanted emails.



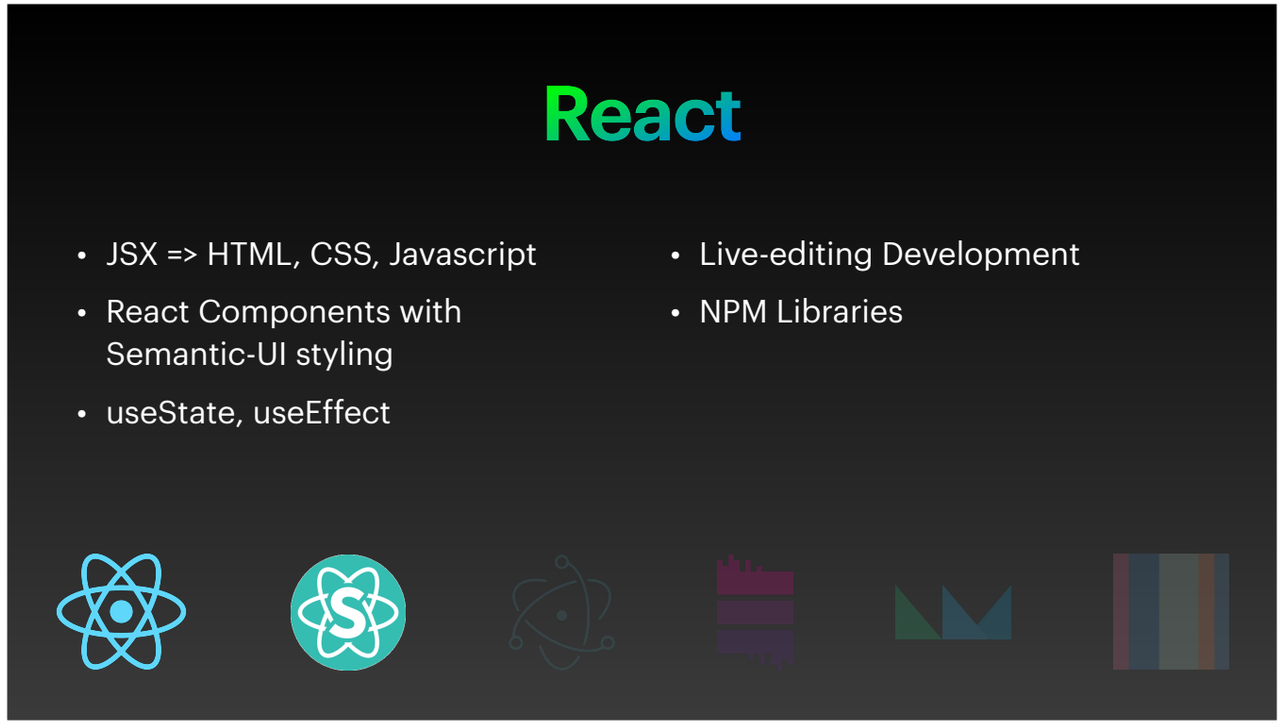
The idea behind this project was to create a client application that could identify messages that might contain words and phrases related to spam.
You might recognize some of these icons, but I’m going to talk about how I created a solution with the technologies.
But first, Here is a flowchart.
* HTML in message body may load internet resources.



The application takes two inputs: Email message + Search Words =>
Compute Spam Score
If Spam
    Display Spam Score
Display Message
Now that you have the algorithm, how are you going to implement it?

React

Start by focusing on the user interaction.
React is a JavaScript framework created by Facebook that specializes in creating a Reactive user experience.



You create what is called React Components, that they have termed JSX, but it is simply HTML, CSS, Javascript.
Semantic UI is a CSS framework and features a ton of boilerplate components ready to use.
Among the top reasons for using react is the access to NPM libraries and the live-editing development environment.
But, the real magic to React is how it manages the state of your application. (Using useState and useEffect). You just tell React about the variables that might change depending on the user's actions and React will handle updating the view when the state changes. This makes for an extremely easy to code and highly readable developer experience.

```
npx create-react-app your-app-name
cd your-app-name
yarn start
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

ReactDOM.render(
    <App />,
  document.getElementById('root')
);
```

package.json


```
"name": "your-app-name",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.3"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
```


If you are familiar with NodeJS, you can start from a blank package.json file, but perhaps the easiest way to get started is with a template.
This is going to create a package.json like this one.
- dependencies are installed to node_modules folder
- scripts are used to run the application

Execution starts at index.js and composed of ES6 modules.
You call the ReactDOM render method and pass it a single component and tell it where to insert into the DOM. In index.html You only need a single div element with the id, ‘root’.
You can find all of this in the template that was installed.

Components

The first thing you need to do is layout the user interface into components
toolbar
import modal
spam words modal
mailbox
reader
spam score modal

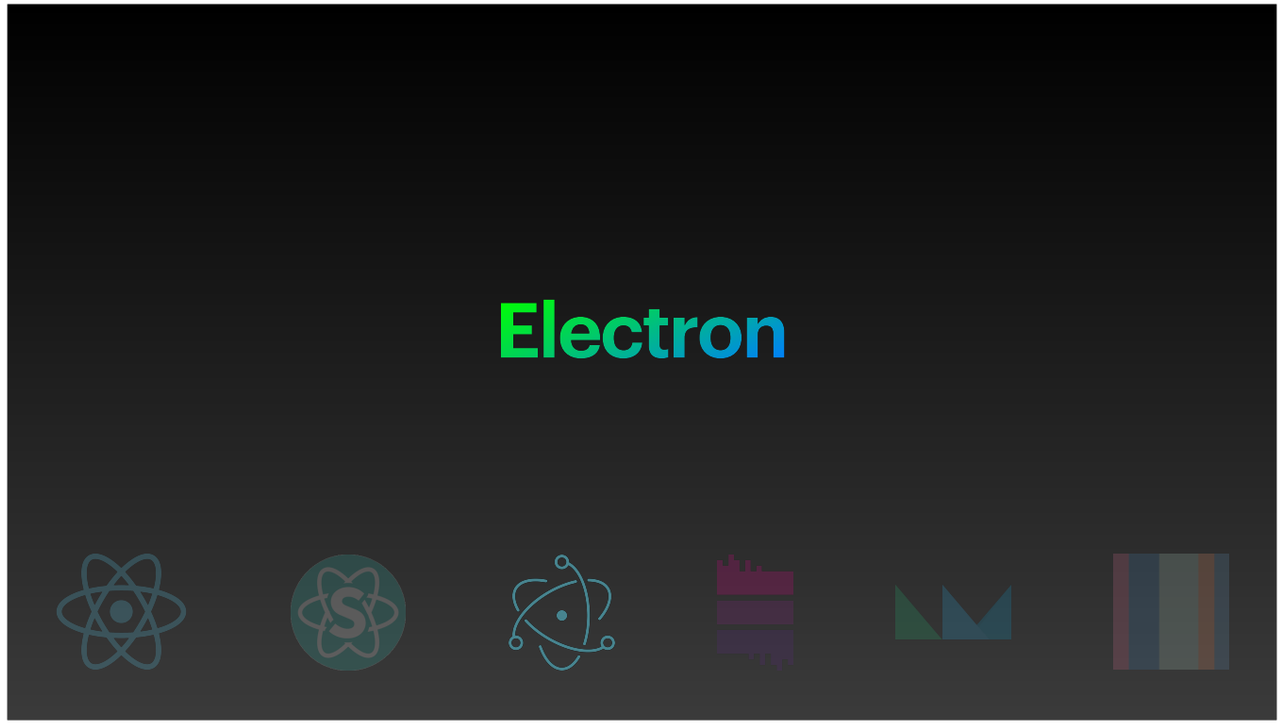
React State

```
const [messages, setMessages] = useState([])
const [spamWords, setSpamWords] = useState([])
const [selectedMessage, setSelectedMessage] = useState({})
```

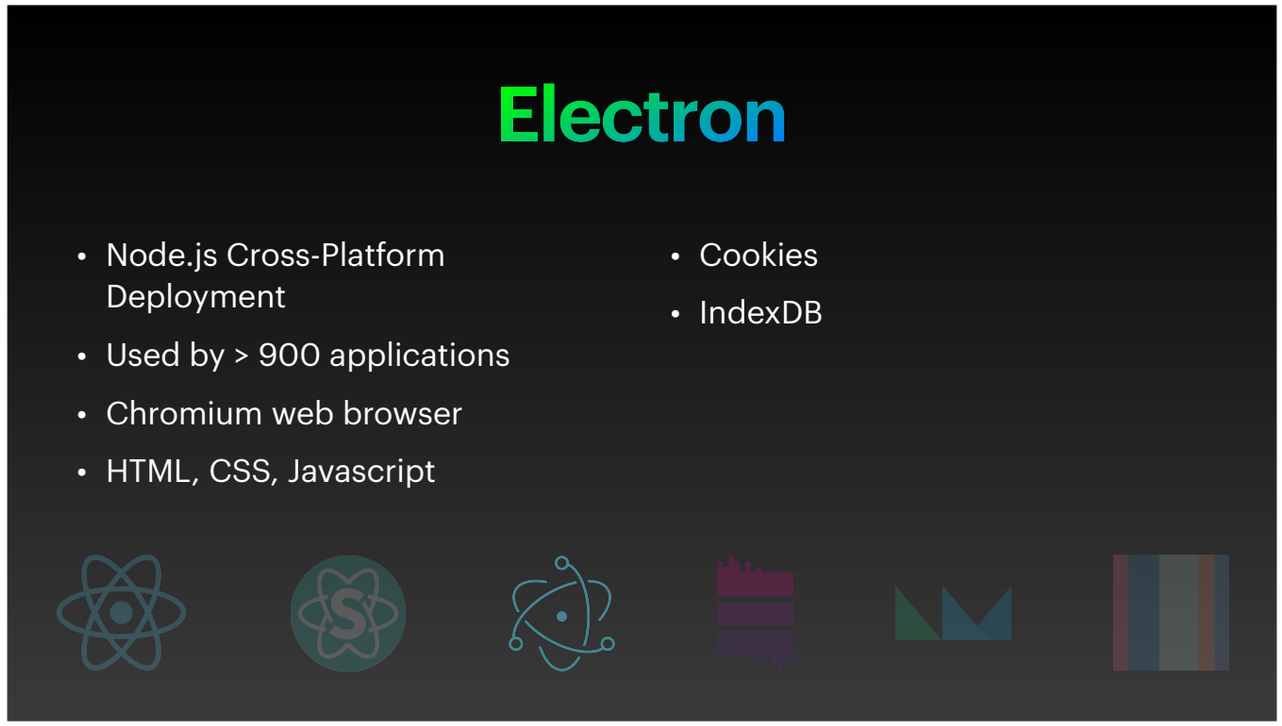
```
useEffect(() => {
  console.log("selected message")
}, [selectedMessage])

useEffect(() => {
  console.log('messages or words have changed in App')
}, [messages, spamWords])
```

State is created in App.jsx and passed down to children as properties.
If the state is modified, the children re-render and everything appears up-to-date.



Users normally interact with a website with a web browser of their choice by browsing to a url in the form of www.domain.com. Users can typically make a distinction between applications running on their computer versus browsing to a website. Electron is a Node library that bridges the gap between a web application and a system application. Developers can create an executable that runs like a system application but developed like a website.



Because the app is built on top of a version of Chromium, the developer can take advantage of all the resources of traditional web development. There’s not much you have to do other than link up your options in package.json to load an html file.

package.json

```
"devDependencies": {
  "concurrently": "^5.3.0",
  "electron": "^9.2.1",
  "electron-builder": "^22.8.0",
  "wait-on": "^5.2.0"
},
"build": {
  "productName": "SpamScanner",
  "appId": "com.biesser.SpamScanner",
  "files": [
    "build/**/*",
    "node_modules/**/*"
  ],
  "directories": {
    "buildResources": "assets"
  },
  "mac": {
    "icon": "public/icon.png",
    "hardenedRuntime": true,
    "gatekeeperAssess": false
  },
```

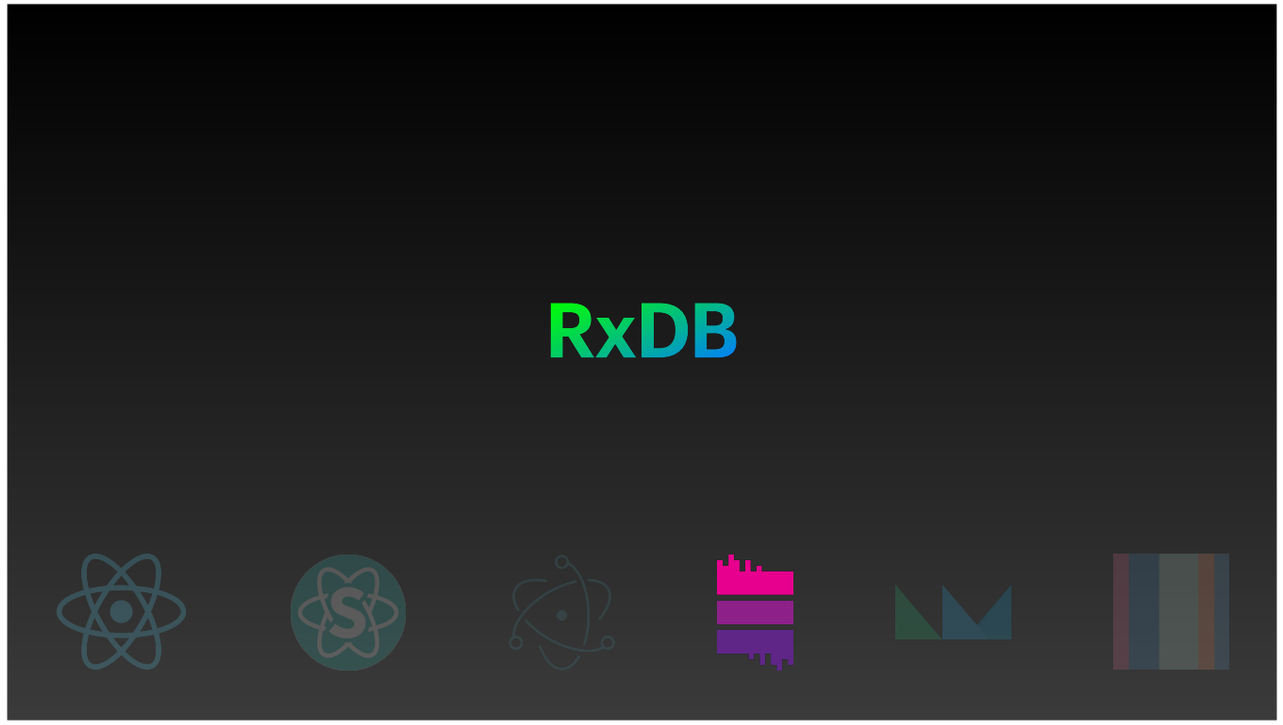
```
"dmg": {
  "icon": "public/icon.png",
  "contents": [
    {
      "x": 134,
      "y": 190
    },
    {
      "x": 344,
      "y": 190,
      "type": "link",
      "path": "/Applications"
    }
  ]
},
"win": {
  "icon": "public/favicon.ico",
  "target": [
    {
      "target": "nsis",
      "arch": [
        "x64",
        "ia32"
      ]
    }
  ]
}
```

add the electron packages
set build options

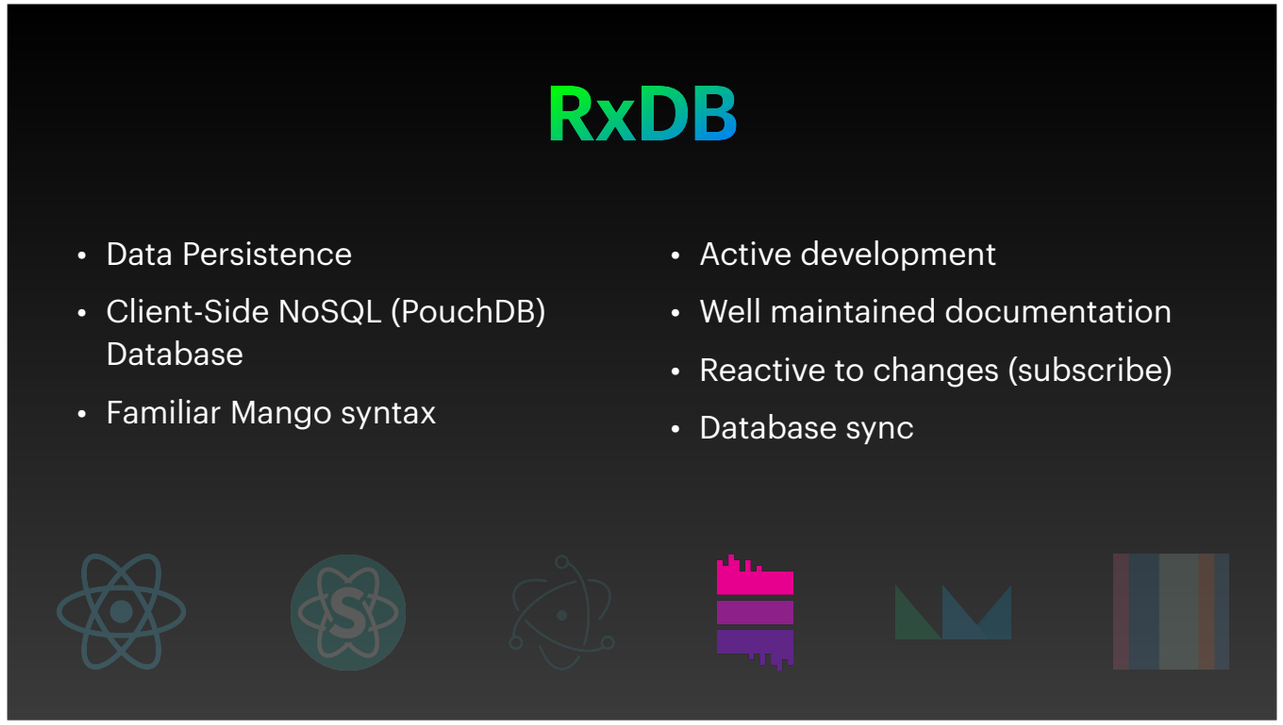
yarn package

```
"scripts": {
  "react-start": "react-scripts start",
  "react-build": "react-scripts build",
  "react-test": "react-scripts test --env=jsdom",
  "react-eject": "react-scripts eject",
  "electron-build": "electron-builder",
  "release": "yarn react-build && electron-builder --publish=always",
  "build": "yarn react-build && yarn electron-build",
  "start": "concurrently \"cross-env BROWSER=none yarn react-start\" \"wait-on http://localhost:3000 && electron .\"",
  "package": "yarn run build && electron-builder build --publish never"
},
```

modify the scripts
packaging the executable is as easy as running yarn package



Next up is how the database will be represented within the application. RxDB is a client side database that can be used for data persistence.



There are many database options out there, but these are some of the reasons RxDB caught my attention.

- Schema
  - Message
  - SpamWord
- Collection
- Documents

```
async function createDatabase(name, adapter) {
  const db = await createRxDatabase({
    name: name,                   // <- name
    adapter: adapter,             // <- storage-adapter
    password: 'myPassword',       // <- password (optional)
    multiInstance: true,          // <- multiInstance (optional, default: true)
    eventReduce: false            // <- eventReduce (optional, default: true)
  });

  console.log('creating collection..');
  await db.collection({
    name: 'messages',
    schema: messageSchema
  });

  console.log('creating collection..');
  await db.collection({
    name: 'spamwords',
    schema: spamWordsSchema,
    statics: {
      // a Promise
      countAllDocuments: async function () {
        const allDocs = await this.find().exec();
        return allDocs.length;
      }
    },
    methods: {}
  });
```

2 Schemas - Message and SpamWord

- define the data the will be stored
- gives structure to a NoSQL database
- string, number, array, object

CRUD

```
db.spamwords.upsert({
  value: selectedWord.toLowerCase(),
  weight: selectedWeight
})
```

```
await db.spamwords.find()
  .exec()
  .then(async (spamWords) => {
    messages.map((message, index) => {
      messages[index].spamScanner = RunSpamScanner(message, spamWords)
      console.dir(message)
    })

    await db.messages.bulkInsert(messages)

  })
```

```
await db.messages.find()
  .exec()
  .then(async (messages) => {
    // RxDB doesn't have a bulk update and likely never will,
    // it is following PouchDB releases
    // instead left with updating messages one at a time.
    messages.map((message, index) => {
      message.update({
        $set: {
          spamScanner: RunSpamScanner(message, spamWords)
        }
      })
    })
  })
```

```
function removeWord(spamWord) {
  console.log(spamWord)
  // call the RxDB document method
  spamWord.remove()
}
```

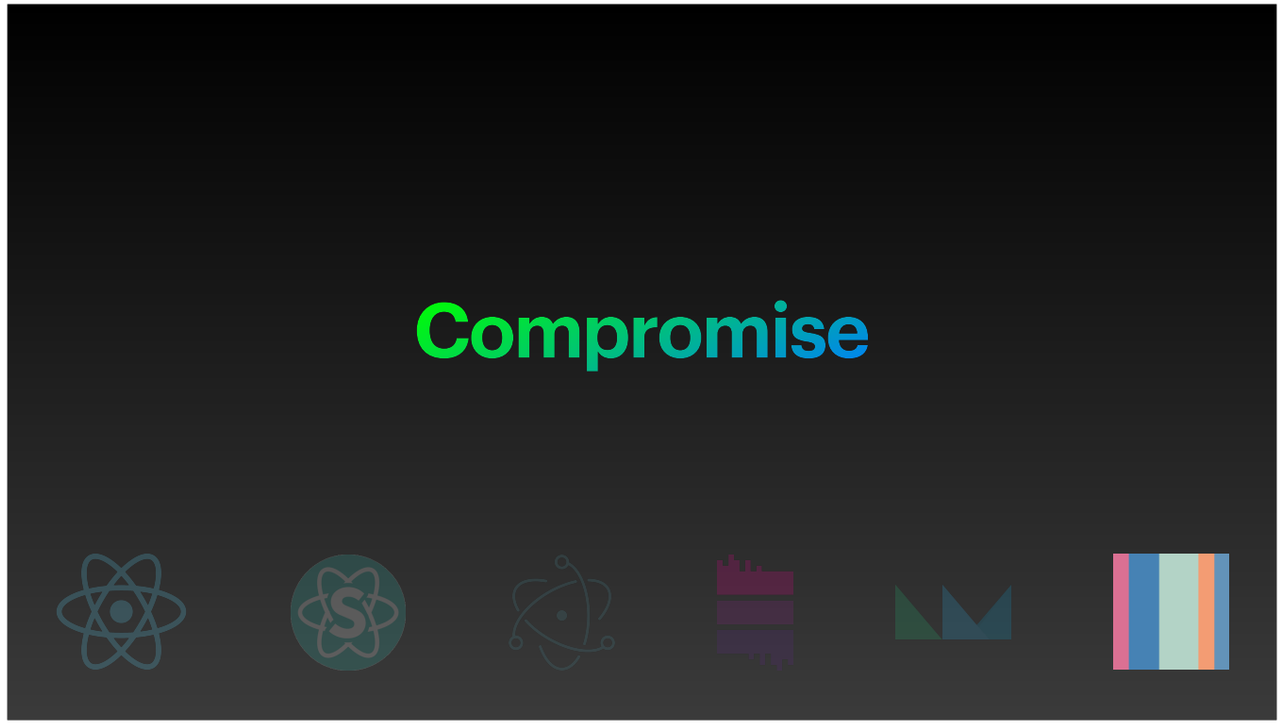
upsert()
find()
bulkInsert()
update()
remove()

The application needed someone of getting messages into the database. This could have been a simple textbox, but why not try to fit a more usable scenario? Ultimately, you might want to connect to the mail server directly, but this solution assumes the user already has access to a mailbox in the mbox format.

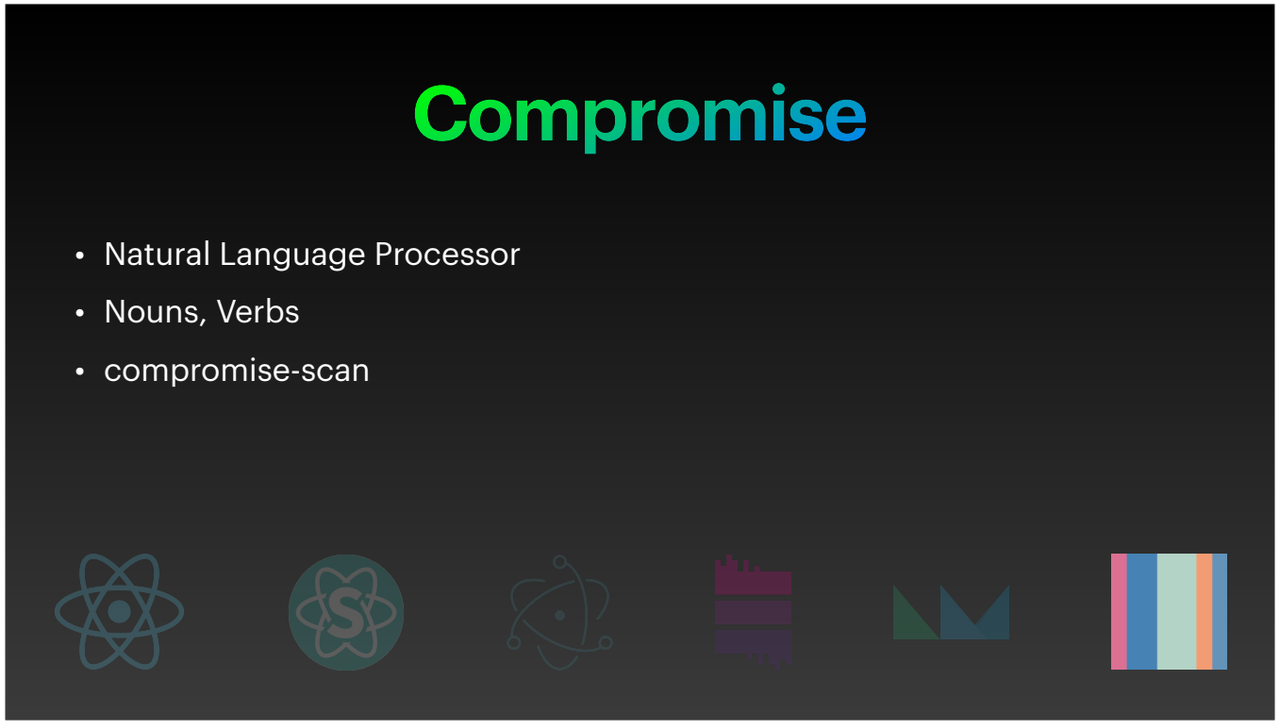
Mailparser

- mbox => JSON
- “From “
- simpleParser(string)

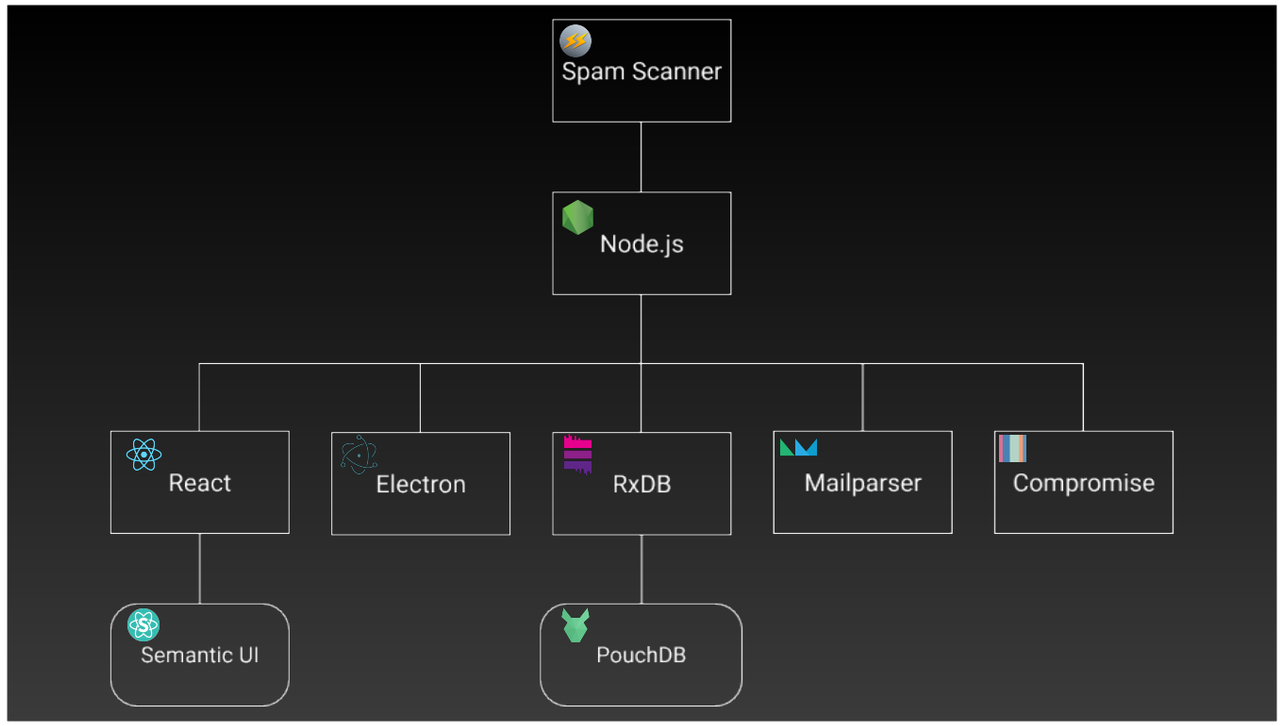
An mbox is nothing more than a text file with messages separated by a line beginning with “From “. It’s rudimentary at best, but has been used since the existence of email. A specially crafted email would break this format. Mailparser is actually a larger library of mail functions and simpleParser is just one method.



There is actually a large field dedicated to researching language using machine learning.



Compromise can take a string and break it down into the individual parts of speech and determines nouns, verbs, subjects, etc. Compromise-scan is a plugin and seemed to be just what I was looking for. You supply a string and an array of words to search for and it will return the number of occurrences.



Dependency graph

Import mbox
Manage Spammy words
Read messages
View Spam Score

References

- React. https://reactjs.org/docs/create-a-new-react-app.html
- Semantic-UI React. https://react.semantic-ui.com/
- Electron. https://www.electronjs.org/docs/tutorial/first-app
- RxDB. https://rxdb.info/
- Mailparser. http://nodemailer.com/extras/mailparser/
- Compromise. http://compromise.cool/