How I created a wedding RSVP app using React, styled-components, and Firebase

Stephen McLean Follow

Jun 6, 2019 · 9 min read ★

Photo by Marvin Meyer on Unsplash

In my last post, I discussed designing an application to allow users to RSVP for a wedding. In this post, I am going to walk through the development of the project, from the initial set-up right through to converting the designs into real components and pages.

Project Set Up

I created the initial project scaffolding using Create React App. Once I had the scaffolding in place, the next step was to install the necessary dependencies.

Font Awesome

Font Awesome provides a fantastic library of free (and paid) icons. I installed their React bindings for use in the project.

```
npm i --save @fortawesome/fontawesome-svg-core \
npm i --save @fortawesome/free-solid-svg-icons \
npm i --save @fortawesome/react-fontawesome
```

Then I created a file to handle registering the icons needed for the project.

src/fontawesome/index.js — Font Awesome initialization

Firebase

I decided to use Firebase for both its database and for hosting. It provides a great API that makes it easy to get your app deployed with a database quickly.

First, I needed to install the dependencies:

```
npm install --save firebase
npm install --save-dev firebase-tools
```

Next, I went through the set-up process using the Firebase CLI and then I was ready to start using Firebase in the project. I created a common file to initialize Firebase and expose the parts of the API I needed.

src/firebase/index.js — Firebase initialization

The Firebase related variables will be consumed from the environment variables, both on CI and locally. Locally, I could then create a `.env` file, or set the variables in the terminal.

Redux Form

The app was small enough that I didn't feel the need to use Redux for state management. However, I did want to use Redux form for handling user input and validation.

```
npm install --save redux react-redux redux-form
```

Once the dependencies were installed I added a file to reference the form reducer.

src/reducers/index.js — Reducer creation

And then I wired up the reducer with the store.

src/index.js — Wiring up the reducer with the store.

ESLint, Prettier, and Husky

Create React App comes with ESLint out of the box. I decided to add Prettier for formatting and Husky to ensure the formatting is checked before each commit.

```
npm install --save-dev prettier eslint-plugin-prettier pretty-quick
husky
```

Once installed I added the Prettier plugin to the ESLint config:

```
// /.eslintrc

{
  "extends": "react-app",
  "plugins": ["prettier"],
  "rules": {
    "prettier/prettier": "error"
  }
}
```

And then added the Husky configuration for the pre-commit hook to the `package.json` file:

```
// /package.json

{
...
  "husky": {
    "hooks": {
      "pre-commit": "pretty-quick --staged"
    }
  }
}
```

Other dependencies

The remaining dependencies to add are styled-components for styling, PropTypes for prop-type validation, and React Router for routing.

```
npm install --save styled-components
npm install --save prop-types
npm install --save react-router-dom
```

Continuous Integration

With the project set up, I was ready to add CI. I chose CircleCI as it integrates nicely with GitHub and is easy to use.

In addition to installing dependencies and building the app, I also use CircleCI to deploy the app to Firebase. I use feature branching on most of my projects so I only needed `master` to be deployed.

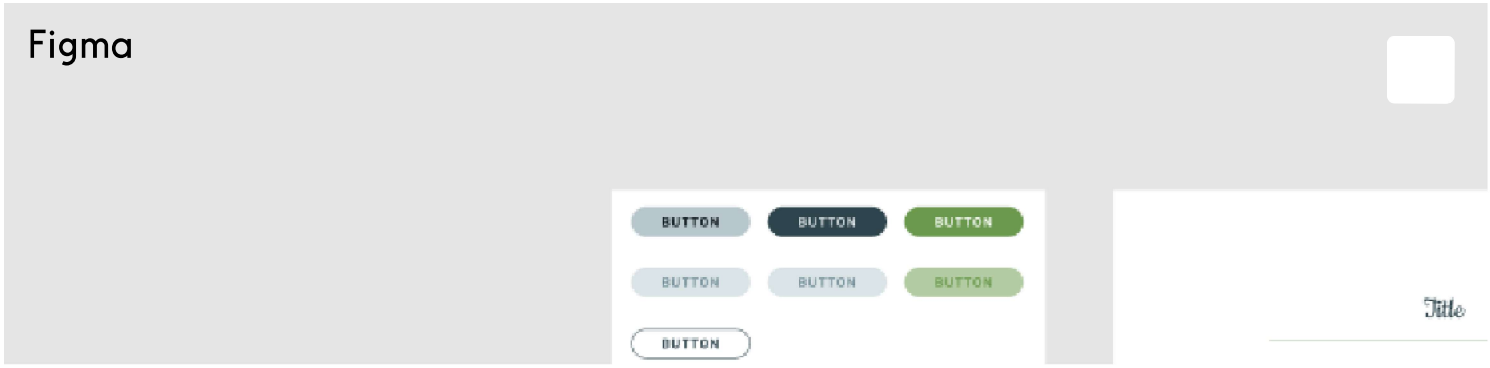
CircleCI configuration

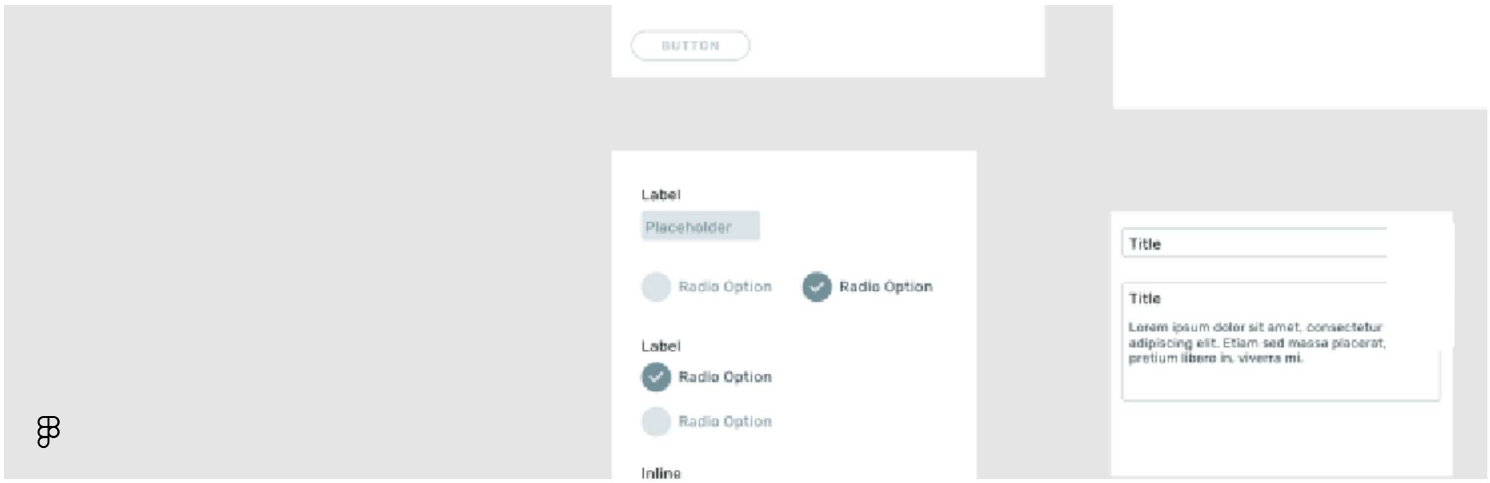
On line 55 above you can see the command to deploy to Firebase, using an environment variable to store the authentication token. The `firebase-tools` dependency we installed earlier means that we don't have to install a separate dependency for CI, we just use the one that comes with the build.

On line 65–67 you can see where I have restricted the deployment job to run only on the `master` branch.

Components

With everything set up, it was time for the real work to begin 😊. First, let's take a look at the designs to see what I was aiming for.





Component mock-ups

Button

Source of the Button component. Helper functions and imports omitted.

- You can see that our Button component has very little functional logic, except for the styling helper functions. I have omitted them here, but they essentially just use the `buttonType` and `buttonStyle` props to lookup the correct colors and border style to use.
- The only other thing to note here is that we set the `type` of every button to `button` by default. This is to avoid any issues later down the line when we eventually have multiple buttons inside a Redux Form. Any button that is not intended to submit the form should have its type set correctly.

Input

Source for the Input component. Imports and exports omitted.

- Our Input component is made up of an optional label and then a styled HTML input.
- On line 25 you can see the expected props for the Input component. One to note is the `input` prop. This is here because of Redux Form. This prop will get passed by Redux Form and will contain the `value` and `onChange` props you would normally expect to find on a component like this.

Radio

To handle radio inputs I created two components. `Radio` , would represent a *single* radio button. `RadioGroup` would take a list of options and render a `Radio` for each option.

- The basic structure and styling of the solution has been taken from this great guide by W3 Schools.
- We hide the regular `input` since it's difficult to style, but we still use it for handling the data.
- Then, based on the `input` checked status, we change the visibility and styling of the surrounding components.

Next, let's look at the `RadioGroup` component and how we reuse the `Radio` component to achieve the final goal.

I have omitted everything except the core logic from this snippet. I think it really shows how simple components can be in React once you break everything down into logical chunks.

Storybook

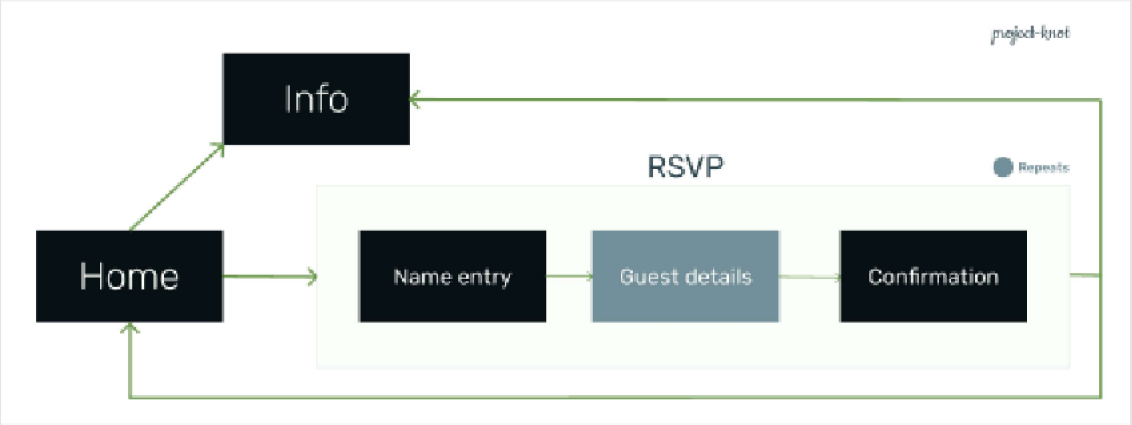
All of the components used in the application, including those I have left out here, can be found on this Storybook instance. The full source code can be found at the end of this article.

Routing

The next step was to start using our new components to put together the pages. Before that, I needed to set up the routing for the project.

Here is the flow diagram I was basing the pages off:

Figma



project-knot-mockups Edited 10 months ago

With that in mind, I created a file to store the routing structure.

src/routes/routes.js — Route definitions

This gave me an easy way to reference routes from different components, without having to rely on hard-coding URL strings.

Next, I used the route definitions along with `react-router` to put everything together.

A note on the app entry file

It would have been much cleaner to export the routes as a list from the route definition file and iterate them here, rather than defining them individually. I've had it on the backlog for a while but I haven't managed to get around to it. If you're implementing something similar consider doing it the cleaner way instead 😊.

Pages

With the routing in place, things were starting to take shape. You might have noticed some of the components I referenced in the `/pages` directory when I was defining the routes above. While setting up the routing I had stubbed those components. Now was the time to fill those out.

Home Page

Figma

project-knot-mockups Edited 10 months ago

What I was aiming to create for the home page.

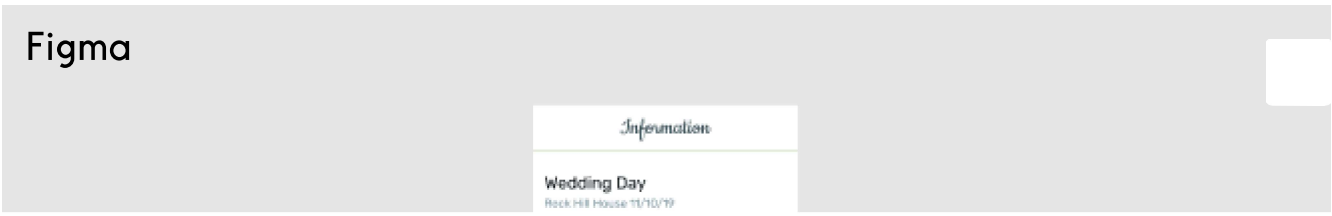
The home page is pretty simple, and there's no real logic to it other than directing to the other pages. Here is what I came up with (imports/exports omitted for the sake of brevity):

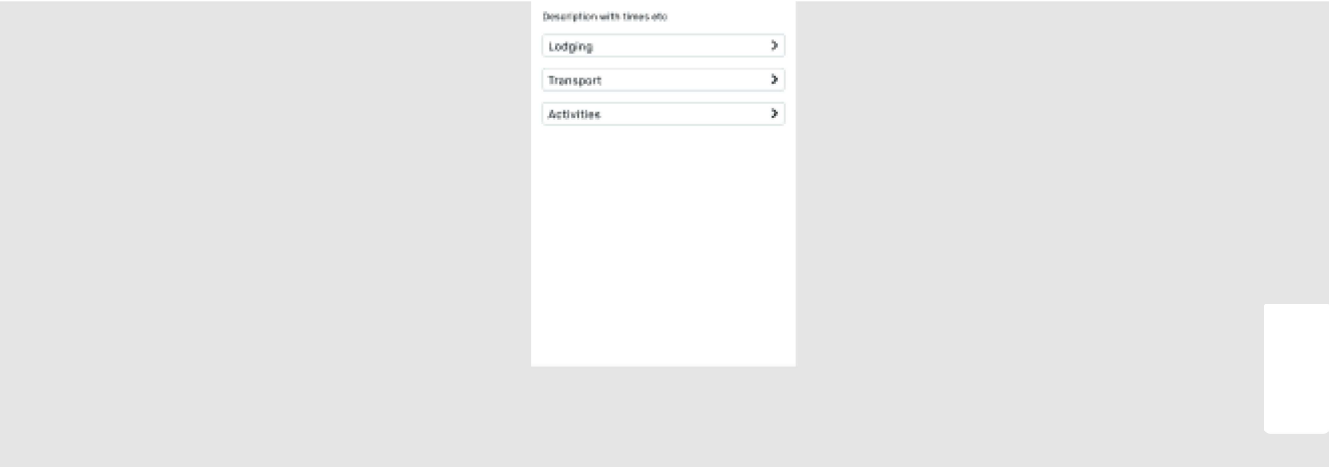
Home page implementation

There's not a huge amount to note here. However, I would like to point out how having a central place for routing makes it really easy to use throughout your application.

Secondly, you may have noticed that I defined three separate functions for handling the three different button clicks. I could have defined a single function here and passed an argument instead. However, defining them separately means I don't have to do any extra work binding, and I also think it makes the code more extensible in the future. Let's say that for some reason navigating to the RSVP page involved some extra work at some point in the future. With one function I'd either have to split it out or make it conditional. Whereas with the current approach I can just make the changes inline to the existing function.

Information Page





project-knot-mockups Edited 10 months ago

The design for the information page

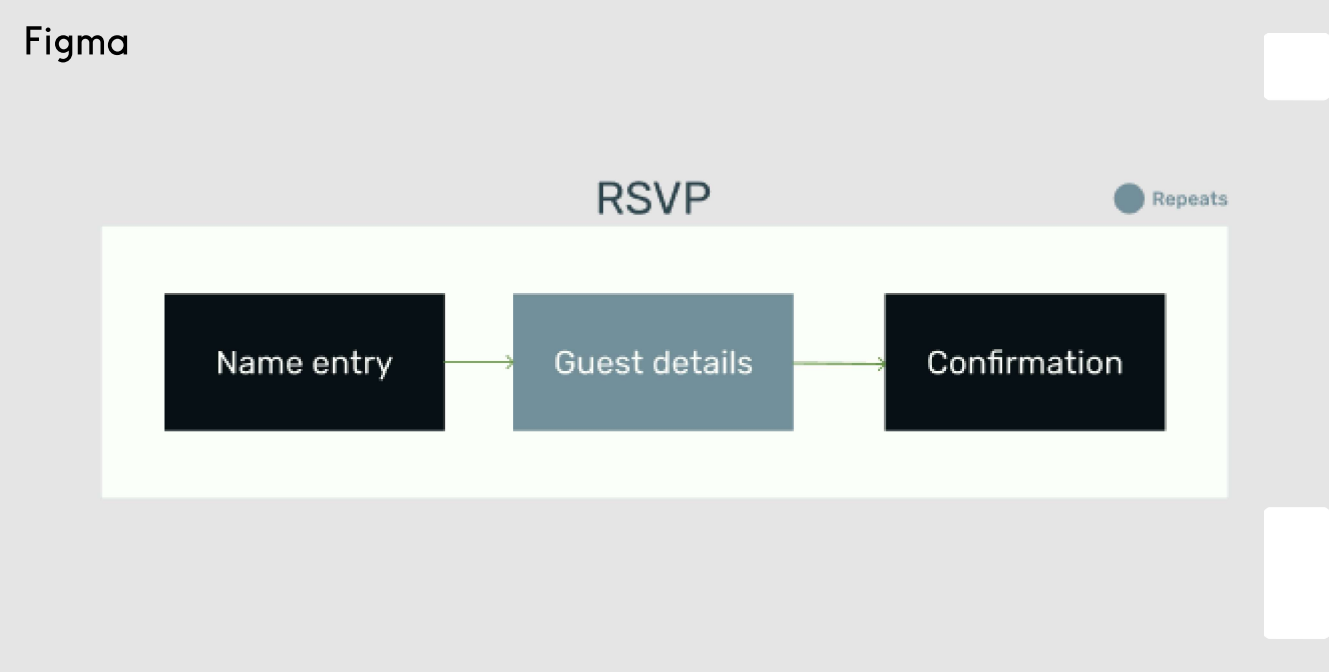
Similar to the home page, the information page is quite straight forward. Its intention is to provide the user with some read-only information about the event.

By utilizing the components we created above, here is what I created (imports omitted):

Information page implementation

RSVP Flow

Finally, I was ready to start on the most complex part of the application. First, let's remind ourselves of what the RSVP flow should look like.

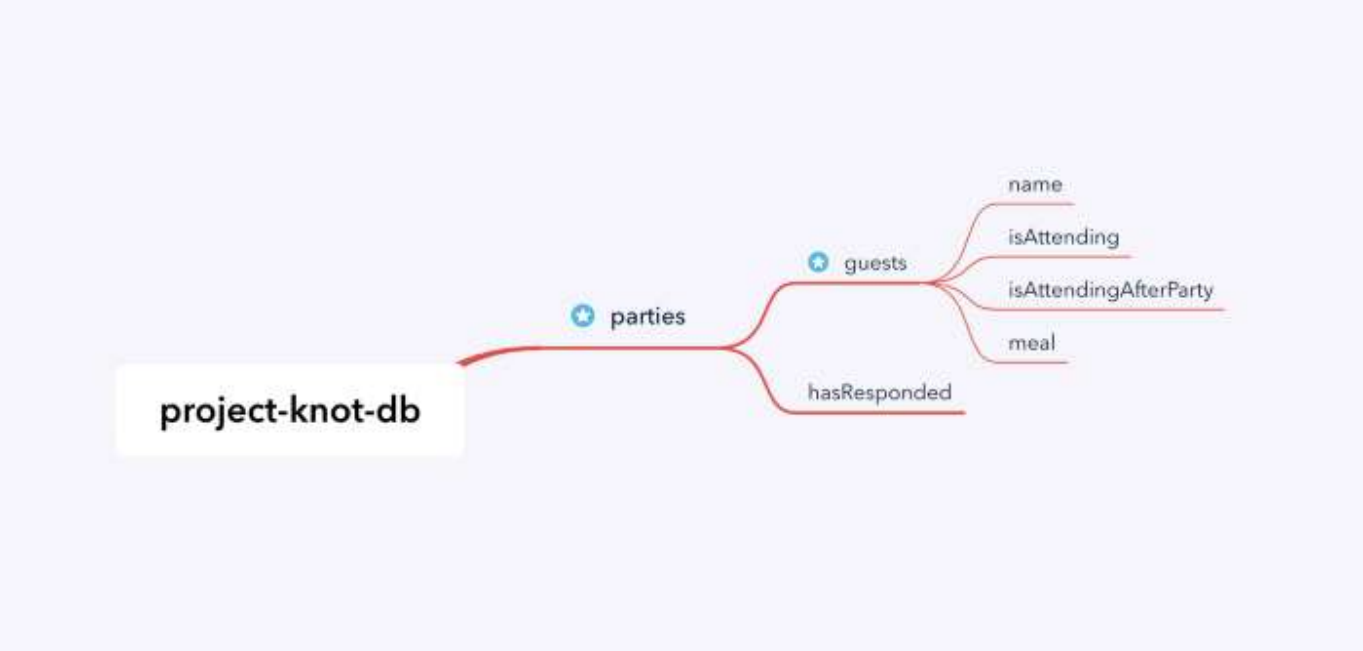


Flow diagram for the RSVP process

Database

Before we start looking at the code for this flow, however, we first need to understand how the data for the RSVP flow will be represented.

The little blue stars in the diagram below represent ***arrays.***



- Data for the guests is stored in Firebase's Real-time Database.
- Our database is made up of a list of *parties,* which in turn are made up of lists of *guests*.
- Each party has a `hasResponded` boolean which is used to denote if the entries have been made for that particular party i.e has the RSVP flow already been completed.

Now we know how the data is represented in the database, how do we use it?

1. When a user enters his/her name on the first step, we query the database for a `party` with a matching guest.
2. We retrieve that party and use the `guests` list to populate the `Guest Details` step(s).
3. When the **last** `Guest Details` step has been completed, we then update the `party` from step 1 with the newly entered details.

RSVP component code. Imports/exports omitted

The code above is for the main RSVP component, with some of the more trivial functions truncated for the sake of readability here. Essentially it's the wrapper for the entire flow. It is responsible for:

1. Loading the list of parties from the database.
2. Loading the appropriate party based on the guest's name.
3. Writing the update party back to the database.
4. Controlling what part of the flow is rendered based on the state.

A lesson learned from structuring the flow as shown in the code above:

From an early point, I decided to structure the RSVP flow using a main parent component that would handle quite a lot of the logic. The app I was creating was never

going to grow or be continually updated. It had to serve its purpose and no more. Because of that, I didn't mind so much that the RSVP flow wasn't going to be extendable.

However, I think you can see from the code above, that structuring a wizard flow like this only really works if you have a super simple use case like mine. If my wizard was to gain another page it would add some more complexity to a component that was already starting to get pretty bloated.

If I was creating an application in the future that required a wizard, I would definitely go about structuring it a different way. Utilizing `react-router` and having a separate route for each of the pages I think would be a starting point.

And finally, one last piece of code to talk about:

The last thing I'd like to show is how I handled the guest details steps. For each guest in a party, the user is required to fill out a form detailing that guest's choices. It's the same form each time, but for a different guest. The user can also navigate forward and backward between different guests.

I handled this using two components. `GuestsForm` was responsible for deciding which form to render, and switching between them. And `SingleGuestForm` was responsible for rendering the actual form elements for a single guest.

The code above shows the logic in `GuestsForm` for rendering a single form. Of note here is the `initialValues` variable that we pass to the `SingleGuestForm` component on line 12. This variable is there because of the need to be able to go back to a previously filled out form. The `SingleGuestForm` will pre-populate the form with those values if available.

And lastly, here is some code from my `SingleGuestForm` component. Of note here is line 3, where I enable reinitialization for the form. You can read more about the different configuration values available on Redux Form here.

Conclusion and source code

I hope you have learned something from my walkthrough of this project. If you have any feedback or suggestions, I would love to hear them in the responses.

The source code for this project can be found here.

While you are here feel free to look at some of my other articles on development/design 👇. Thank you very much for reading!

developer

It's been almost four years since I graduated with a degree in CS and began my career as a Software Developer. In this...

medium.freecodecamp.org

Sign up for Top 10 Stories

By The Startup

Get smarter at building your thing. Subscribe to receive The Startup's top 10 most read stories — delivered straight into your inbox, twice a month. Take a look.

Get this newsletter

Emails will be sent to sonnsonn06@gmail.com.
Not you?

Software Development React JavaScript Firebase Continuous Integration