Susanna Kline
Technical Writing Portfolio

VA Forms Library — Using form widgets and fields (2022)

Document type
Reference

Audience
Software developers within Veterans Affairs who have a basic understanding of the Forms Library and its schema and uiSchema.

Goals
Based on user research, I chose to prioritize the following items:

- Clearly document how to use each widget
- Show a screen shot or video of each widget when possible, supported by high quality alt text for screen captures and videos
- Show a code example of each
- Link to an example in the VA code base

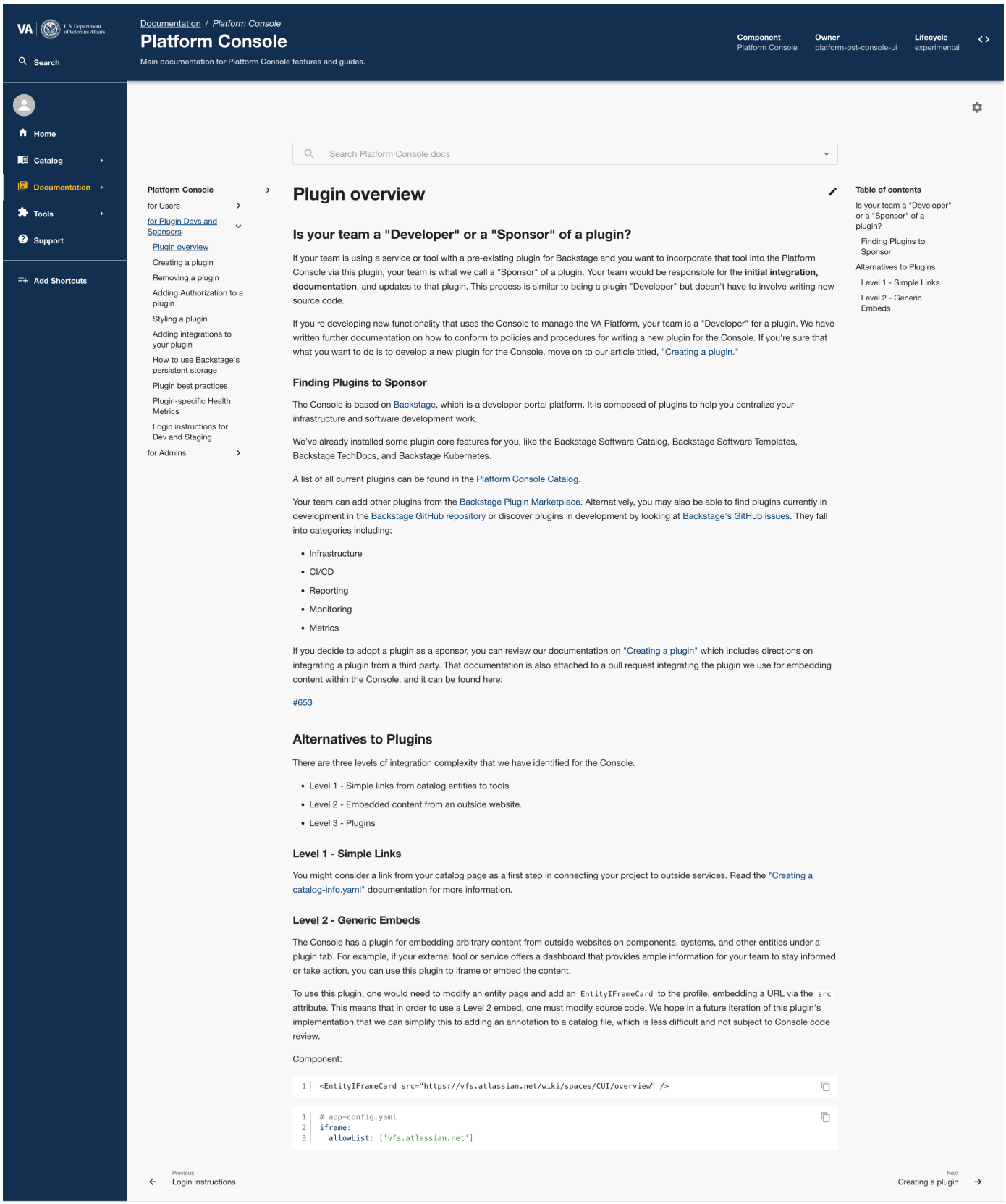
VA Forms Library - Using Form Widgets and Fields (Link to document)

VA Platform Console — Plugin overview (2022)

Document type
Technical overview document of a backstage.io feature integrated with the VA Platform Console

Audience
Software developers within Veterans Affairs who have a basic understanding of the Console.

Goals
- Introduce Console developers to the purpose of plugins
- Explain the purpose of sponsoring plugins
- Provide developers with alternatives to sponsoring plugins.


VA
U.S. Department of Veterans Affairs
Search
Documentation / Platform Console
Platform Console
Main documentation for Platform Console features and guides.
Component
Platform Console
Owner
platform-pst-console-ui
Lifecycle
experimental
Home
Catalog
Documentation
Tools
Support
Add Shortcuts
Search Platform Console docs
Platform Console
for Users
for Plugin Devs and Sponsors
Plugin overview
Creating a plugin
Removing a plugin
Adding Authorization to a plugin
Styling a plugin
Adding integrations to your plugin
How to use Backstage's persistent storage
Plugin best practices
Plugin-specific Health Metrics
Login instructions for Dev and Staging
for Admins
Plugin overview
Is your team a "Developer" or a "Sponsor" of a plugin?
Finding Plugins to Sponsor
Alternatives to Plugins
Level 1 - Simple Links
Level 2 - Generic Embeds
Table of contents
Is your team a "Developer" or a "Sponsor" of a plugin?
If your team is using a service or tool with a pre-existing plugin for Backstage and you want to incorporate that tool into the Platform Console via this plugin, your team is what we call a "Sponsor" of a plugin. Your team would be responsible for the initial integration, documentation, and updates to that plugin. This process is similar to being a plugin "Developer" but doesn't have to involve writing new source code.
If you're developing new functionality that uses the Console to manage the VA Platform, your team is a "Developer" for a plugin. We have written further documentation on how to conform to policies and procedures for writing a new plugin for the Console. If you're sure that what you want to do is to develop a new plugin for the Console, move on to our article titled, "Creating a plugin."
Finding Plugins to Sponsor
The Console is based on Backstage, which is a developer portal platform. It is composed of plugins to help you centralize your infrastructure and software development work.
We've already installed some plugin core features for you, like the Backstage Software Catalog, Backstage Software Templates, Backstage TechDocs, and Backstage Kubernetes.
A list of all current plugins can be found in the Platform Console Catalog.
Your team can add other plugins from the Backstage Plugin Marketplace. Alternatively, you may also be able to find plugins currently in development in the Backstage GitHub repository or discover plugins in development by looking at Backstage's GitHub issues. They fall into categories including:
Infrastructure
CI/CD
Reporting
Monitoring
Metrics
If you decide to adopt a plugin as a sponsor, you can review our documentation on "Creating a plugin" which includes directions on integrating a plugin from a third party. That documentation is also attached to a pull request integrating the plugin we use for embedding content within the Console, and it can be found here:
#653
Alternatives to Plugins
There are three levels of integration complexity that we have identified for the Console.
Level 1 - Simple links from catalog entities to tools
Level 2 - Embedded content from an outside website.
Level 3 - Plugins
Level 1 - Simple Links
You might consider a link from your catalog page as a first step in connecting your project to outside services. Read the "Creating a catalog-info.yaml" documentation for more information.
Level 2 - Generic Embeds
The Console has a plugin for embedding arbitrary content from outside websites on components, systems, and other entities under a plugin tab. For example, if your external tool or service offers a dashboard that provides ample information for your team to stay informed or take action, you can use this plugin to iframe or embed the content.
To use this plugin, one would need to modify an entity page and add an EntityIFrameCard to the profile, embedding a URL via the src attribute. This means that in order to use a Level 2 embed, one must modify source code. We hope in a future iteration of this plugin's implementation that we can simplify this to adding an annotation to a catalog file, which is less difficult and not subject to Console code review.
Component:
1 <EntityIFrameCard src="https://vfs.atlassian.net/wiki/spaces/CUI/overview" />
1 # app-config.yaml
2 iframe:
3 allowList: ['vfs.atlassian.net']
Previous
Login instructions
Next
Creating a plugin

VA Platform Console — Adding integrations to your plugin (2022)

Document type
API integration

Audience
Software developers within Veterans Affairs who have a basic understanding of the Console and would like to read data from or write data to providers outside the console.

Goals
Introduce Console developers to plugin integrations and instruct them on how to integrate with APIs.

Adding integrations to your plugin

What are integrations?

If you want to read or write data to providers outside the Platform Console, you can configure and use integrations in your plugin.

Some examples of integrations Backstage allows are AWS and Datadog. The list of integrations may change, so check Backstage's Integrations page for all possible options.

Once the integration is configured, either by you or the Platform Console team, you will still need to access, or consume it, using the API. You can do this with a basic fetch request.

Adding a Backstage API

If the API you want to use is provided by Backstage, adding only requires importing it's API reference and retrieving the API with `useApi`.

```
import { useApi, githubAuthApiRef } from '@backstage/core-plugin-api';

const githubAuthApi = useApi(githubAuthApiRef);
```

If you need to add your own API, it must be registered to Backstage using an ApiFactory. See Providing your plugin with your own API below.

Using Platform Console-provided integrations

Currently, the Platform Console is configured for:

- GitHub Auth API

Integrating with the GitHub Auth API

The GitHub Auth integration allows you to authenticate with the GitHub Auth API using this signature:

```
githubAuthApiRef: ApiRef<
  OAuthApi & ProfileInfoApi & BackstageIdentityApi & SessionApi
>;
```

Several other APIs are nested in the signature, and we include details for each of these here.

If you don't need user profile info, or session info, you may only need the OAuth API, which provides your authentication toward GitHub APIs. Then you can use the access token with a fetch request or the `@octokit` request library to make authenticated requests to GitHub's REST API:

```
type OAuthApi = {
  getAccessToken(
    scope?: OAuthScope,
    options?: AuthRequestOptions,
  ): Promise<string>;
};
```

ProfileInfoApi provides access to a user's profile information from an auth provider, in this case, the GitHub profile:

```
type ProfileInfoApi = {
  getProfile(options?: AuthRequestOptions): Promise<ProfileInfo | undefined>;
};
```

BackstageIdentityApi provides access to the user's identity within Backstage

```
type BackstageIdentityApi = {
  getBackstageIdentity(
    options?: AuthRequestOptions,
  ): Promise<BackstageIdentityResponse | undefined>;
};
```

SessionApi provides controls for auth providers tied to a persistent session

```
type SessionApi = {
  signIn(): Promise<void>;
  signOut(): Promise<void>;
  sessionState$(): Observable<SessionState>;
};
```

Providing your plugin with your own API

In your `plugin.tsx` file, your plugin is created with a `createPlugin` call. Here you can provide your plugin with any APIs that it may use. Pass `createApiFactory` an object with three keys:

- `api` - an API reference of the API you want to use
- `deps` - an object of dependencies providing API references
- `factory` - a factory function to return the implementation of the API you want to use For example, the `githubRateLimitCheckPlugin` uses the `GithubRateLimitAPI`. It's passed the `githubRateLimitApiRef` for the requested API, `githubAuthApiRef`, `fetchApiRef` and `discoveryApiRef` as dependencies, and returns a new `GithubRateLimitClient` from its factory function:

```
export const githubRateLimitCheckPlugin = createPlugin({
  id: 'github-rate-limit-check',
  apis: [
    createApiFactory({
      api: githubRateLimitApiRef,
      deps: {
        githubAuthApi: githubAuthApiRef,
        fetchApi: fetchApiRef,
        discoveryApi: discoveryApiRef,
      },
      factory: ({ githubAuthApi, fetchApi, discoveryApi }) =>
        new GithubRateLimitClient({ githubAuthApi, fetchApi, discoveryApi }),
    }),
  ],
  routes: {
    root: rootRouteRef,
  },
});
```

Now that your plugin is being provided with an API, it can be retrieved by your plugin using `useApi` like Backstage provided APIs.

Reference

- Backstage.io - Utility APIs

Thinking in React: Reusing components (2022)

Document type
Tutorial

Audience
Software developers with an understanding of JavaScript and JSX, who can create a simple React App and run the app locally.

Goals
- Teach the high-level concept of reusable components
- Implement a basic, reusable component in React
- Give the audience ideas of other reusable components that they can try next

---

Thinking in React: Reusing components

Prerequisites:
- You know basic JavaScript and JSX
- You can create a simple React App
- You can run the React app on localhost:3000

Today, we're creating an app for our customer, DeliciousDonuts. They want their website to show a menu of current donuts on the front page. Right now, the list includes nine donuts, but this may change later.

For each donut, they want to show these items:
- donut name
- photo
- price
- ingredients
- "Add to Cart" button

For now, we'll think about this app at a high level and work with only minimal formatting to focus on the user interface (UI) functionality. In a later lesson, we'll add CSS to make things look good for the customer.

Once you've created and started your React app, we'll work in a simple `App.js` file and create a header, DeliciousDonuts.

App.js

```
1  import './App.css';
2
3  function App() {
4    return (
5      <div style={{textAlign: 'center'}}>
6        <header>
7          <p>DeliciousDonuts</p>
8        </header>
9      </div>
10   );
11 }
12
13 export default App;
```

If you've read through the React docs, you may know that a big part of the reason for using React is to break your app into reusable components. This means you can reuse code in multiple places without having to rewrite it.

But how do you identify parts of the code where it makes sense to reuse components? Let's look at our DeliciousDonuts app. We need a page that displays nine of the same thing, with a donut name, photo, price, ingredients, and a button.

We know we need nine of something that looks like this to show up on our app page:

Sprinkles donut

Price: $4.50
Ingredients: Sprinkles, milk, yeast, eggs, butter, sugar, flour, salt, oil.
Add to Cart Button

If we need more than one of anything, it's a good place to think about writing a reusable component. But you might be wondering, if each one has a different name, photo, price, and ingredients, how will we reuse the code? That's a great question.

We're going to create a Donut component with the basic structure, but it will have variables that we can replace when we use the component. In React, these variables are called properties, or for short, props.

Let's make a new, reusable `Donut.js` file that we can use as many times as we'd like in our app file. It may not seem like a lot of code to save right now—we could just copy and paste it—but imagine DeliciousDonuts expanded and started selling hundreds of kinds of donuts. We wouldn't want to create new code for each donut!

For now, we'll create this file and include placeholders for everything but the button, but we'll swap them out for the props in a minute.

Donut.js

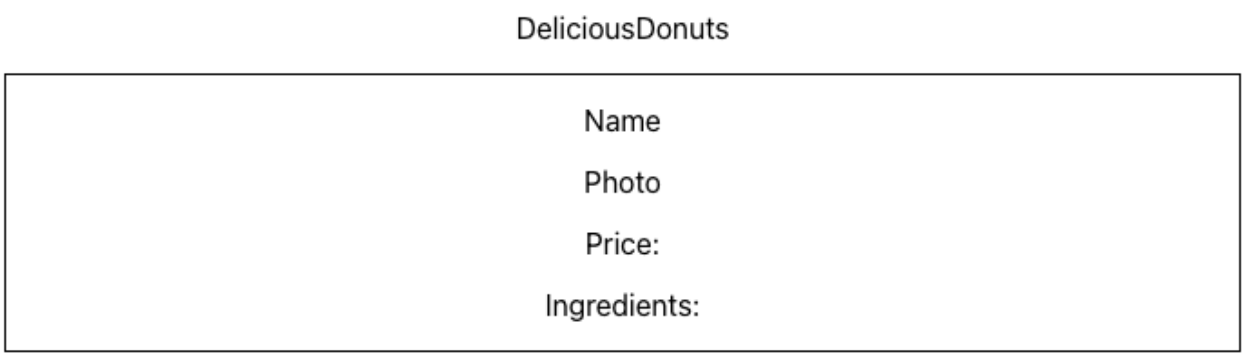
```
1 import './App.css';
2
3 function Donut() {
4   return (
5     <div style={{border: '1px solid black', margin: '4px'}}>
6        <p>Name</p>
7        <p>Photo</p>
8        <p>Price:</p>
9        <p>Ingredients:</p>
10    </div>
11  );
12 }
13
14 export default Donut;
```

Next, we'll import the Donut file back to the `App.js` file (line 2) and insert the component (line 10) below the header:

App.js

```
1 import './App.css';
2 import Donut from './Donut.js';
3
4 function App() {
5   return (
6     <div style={{textAlign: 'center'}}>
7       <header>
8         <p>DeliciousDonuts</p>
9       </header>
10      <Donut />
11    </div>
12  );
13 }
14
15 export default App;
```

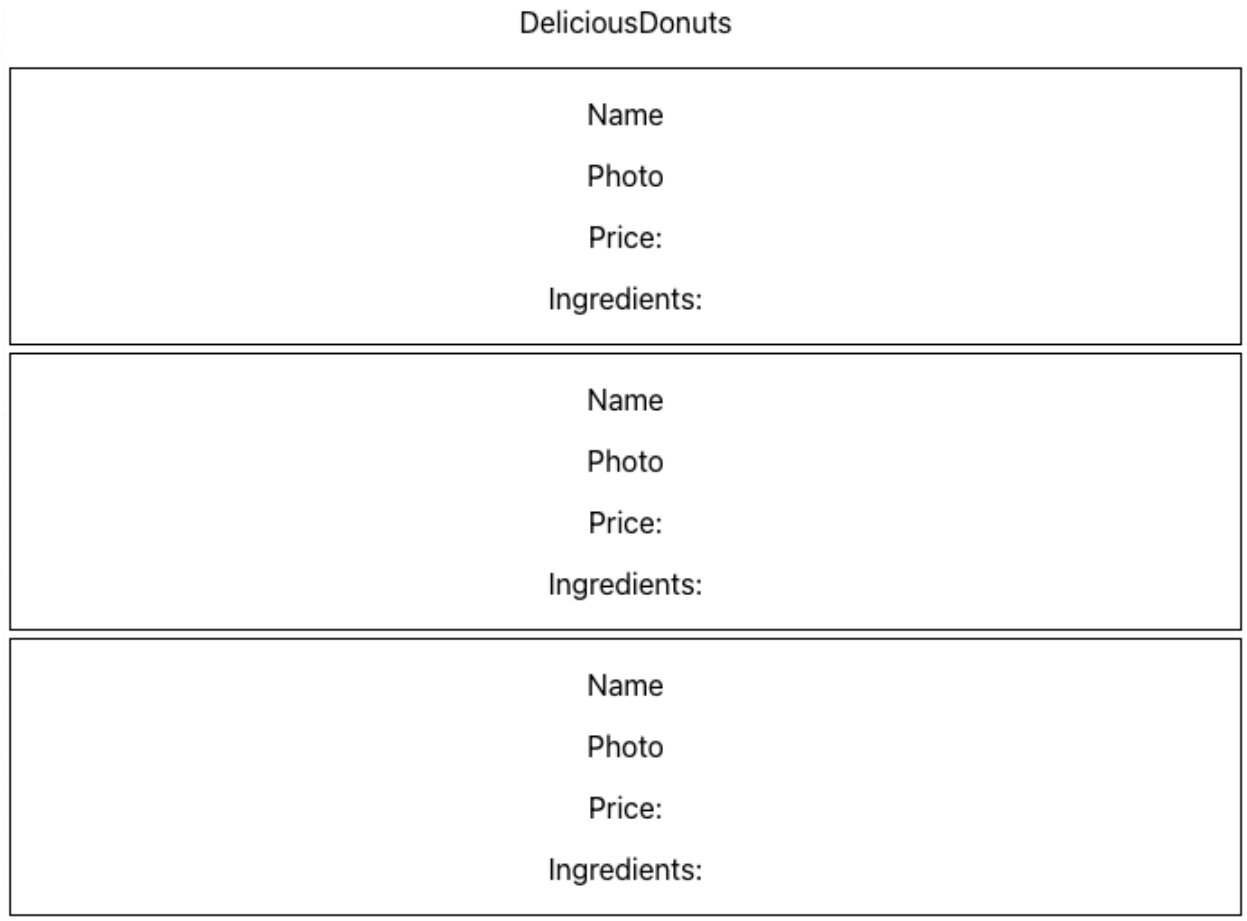
Currently our app looks very simple and shows only our App file and our child component, rendered one time.



We can render it as many times as we want to in our app. We can add two more Donut components to see what it looks like when we render the Donut component multiple times.

App.js

```
1  import './App.css';
2  import Donut from './Donut.js';
3
4  function App() {
5   return (
6     <div style={{textAlign: 'center'}}>
7       <header>
8         <p>DeliciousDonuts</p>
9       </header>
10      <Donut />
11      <Donut />
12      <Donut />
13    </div>
14   );
15  }
16
17  export default App;
```



For now, let's delete those extra two Donut components and add the donut's name prop. This will allow us to list a different donut name every time we render the card.

In the `Donut.js` file, add a property that we can use to pass the donut name in from the app file each time we call our Donut component so it will be unique to each donut. First, in the function (line 3), we add a variable called “donutName.”

Then we’ll replace `<p>Name</p>` (line 6) with `<p>{donutName}</p>`. As a reminder, we need curly braces to tell React that we’re using JSX here instead of html.

Donut.js

```
1  import './App.css';
2
3  function Donut({donutName}) {
4    return (
5      <div style={{border: '1px solid black', margin: '4px'}}>
6          <p>{donutName}</p>
7          <p>Photo</p>
8          <p>Price:</p>
9          <p>Ingredients:</p>
10     </div>
11   );
12 }
13
14 export default Donut;
```

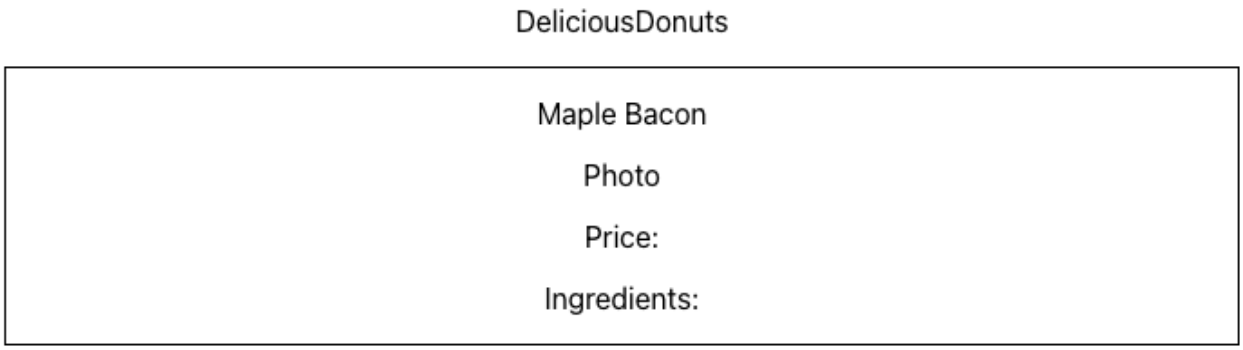
Now that we’ve created a way to insert a Donut name, let’s return to our `App.js` file and insert it.

Instead of `<Donut />` (line 10), now we want to add a prop called `donutName`.

App.js

```
10  <Donut
11        donutName="Maple Bacon"
12  />
```

When you save both files, your React app should look like this.

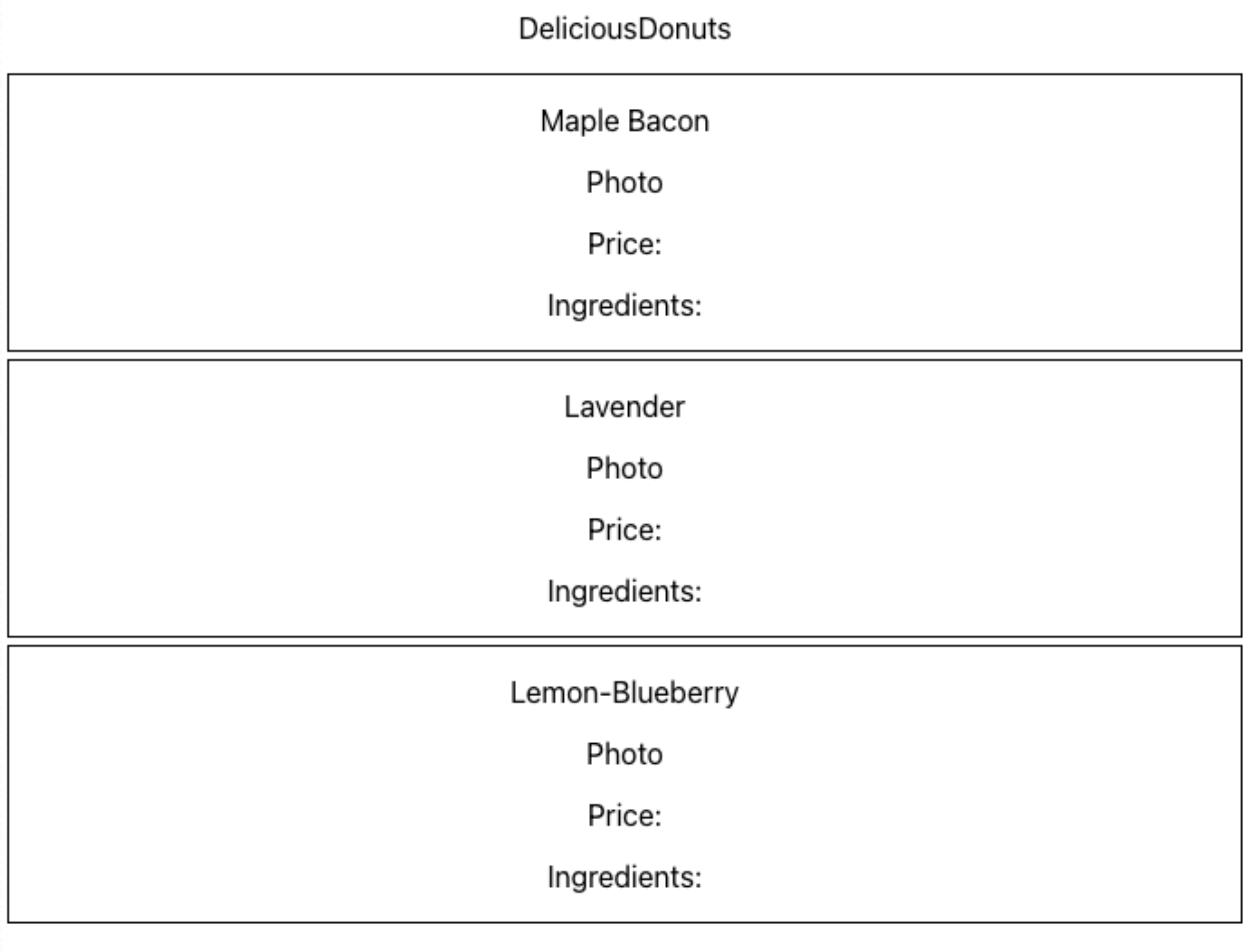


If you want to add more donuts, repeat the Donut component you call in the App file. You can add it as many times as you'd like to add it, each with a different name prop:

App.js

```
1  import './App.css';
2  import Donut from './Donut.js';
3
4  function App() {
5    return (
6      <div style={{textAlign: 'center'}}>
7        <header>
8          <p>DeliciousDonuts</p>
9        </header>
10       <Donut
11         donutName="Maple Bacon"
12       />
13       <Donut
14         donutName="Lavender"
15       />
16       <Donut
17         donutName="Lemon-Blueberry"
18       />
19     </div>
20   );
21 }
22
23 export default App;
```

The corresponding app screen with the Donut components looks like this:



Now you can insert as many properties into the donut card as you'd like following the same pattern.

On your own, try adding the photo, price, and ingredients properties. Then consider other parts of this app or others you've worked in where you can create reusable components, like buttons, menu items, or form fields.

If we zoom out, we can think about the the DeliciousDonuts reusable component concept as a repeated card on a page, and realize that this format applies to many other different kinds of apps: articles or blog posts, photos, book finders, and many others that use a card-like format.

Obviously, the idea of reusable components has immense potential for saving time and allowing you to build more powerful and flexible apps. In the next lesson, we'll implement some other reusable components, including the "Add to Cart" button for our customer, DeliciousDonuts.