Props vs Data in Vue

Vue comes with two different ways of storing variables, **props** and **data**.

These can be confusing at first, since they seem like they do similar things, and it's not clear when to use one vs the other.

So what's the difference between props and data?

`Data` is the private memory of each component where you can store any variables you need. Props are how you pass this data from a parent component down to a child component.

In this article you'll learn:

- What props are, and why **this data only flows down**, not up
- What the `data` option is used for
- What **reactivity** is
- How to avoid **naming collisions** between props and `data`
- How to use **props and data together** for fun and profit 💰

What are props?

In Vue, props (or properties), are the way that we pass data from a parent component down to it's child components.

When we build our applications out of components, we end up building a data structure called a tree. Similar to a family tree, you have:

- parents
- children
- ancestors
- and descendants

Data flows down this tree from the root component, the one at the very top. Sort of like how genetics are passed down from one generation to the next, **parent components pass props down to their children**.

In Vue we add props to components in the `<template>` section of our code:

```
<template>
  <my-component cool-prop="hello world"></my-component>
</template>
```

In this example, we are passing the prop `cool-prop` a value of `"hello world"`. We will be able to access this value from inside of `my-component`.

However, when we access props from inside of a component, we don't own them, **so we can't change them** (just like you can't change the genes your parents gave you).

> *Note: While it's possible to change properties in a component, it's a really bad idea. You end up changing the value that the parent is using as well, which can cause lots of confusion.*

But if we can't change variables, we're kind of stuck.

This is where `data` comes in!

What is data?

Data is the *memory* of each component. This is where you would store data (hence the name), and any other variables you want to track.

If we were building a counter app, we would need to keep track of the count, so we would add a `count` to our `data`:

```
<template>
  <div>
    {{ count }}
    <button @click="increment">+</button>
    <button @click="decrement">-</button>
  </div>
</template>
```

```
export default {
  name: 'Counter',
  data() {
    return {
      // Initialized to zero to begin
      count: 0,
    }
  },
  methods: {
    increment() {
      this.count += 1;
    },
    decrement() {
      this.count -= 1;
    }
  }
}
```

This data is private, and **only for the component itself** to use. Other components do not have access to it.

> *Note: Again, it is possible for other components to access this data, but for the same reasons, it's a really bad idea to do this!*

If you need to pass data to a component, you can use props to pass data down the tree (to child components), or events to pass data up the tree (to parent components).

Props and data are both reactive

With Vue you don't need to think all that much about when the component will update itself and render new changes to the screen.

This is because Vue is reactive.

Instead of calling `setState` every time you want to change something, you just change the thing! As long as you're updating a **reactive property** (props, computed props, and anything in `data`), Vue knows to watch for when it changes.

Going back to our counter app, let's take a closer look at our methods:

```
methods: {
  increment() {
    this.count += 1;
  },
  decrement() {
    this.count -= 1;
  }
}
```

All we have to do is update `count`, and Vue detects this change. It then re-renders our app with the new value!

Vue's reactivity system has a lot more nuance to it, and I believe it's really important to understand it well if you want to be highly productive with Vue. Here are some more things to learn about Vue's reactivity system if you want to dive deeper.

Avoiding naming collisions

There is another great thing that Vue does that makes developing *just a little bit nicer*.

Let's define some props and data on a component:

```
export default {
  props: ['propA', 'propB'],
  data() {
    return {
      dataA: 'hello',
      dataB: 'world',
    };
  },
};
```

If we wanted to access them inside of a method, we don't have to do `this.props.propA` or `this.data.dataA`. Vue let's us omit `props` and `data` completely, leaving us with cleaner code.

We can access them using `this.propA` or `this.dataA`:

```
methods: {
  coolMethod() {
    // Access a prop
    console.log(this.propA);

    // Access our data
    console.log(this.dataA);
  }
}
```

Because of this, if we accidentally use the same name in both our `props` and our `data`, we can run into issues.

Vue will give you a warning if this happens, because it doesn't know which one you wanted to access!

```
export default {
  props: ['secret'],
  data() {
    return {
      secret: '1234',
    };
  },
  methods: {
    printSecret() {
      // Which one do we want?
      console.log(this.secret);
    }
  }
};
```

The real magic of using Vue happens when you start using props and `data` together.

Using props and data together

Now that we've seen how props and data are different, let's see why **we need both of them**, by building a basic app.

Let's say we are building a social network and we're working on the profile page. We've built out a few things already, but now we have to add the

contact info of the user.

We'll display this info using a component called `ContactInfo`:

```
// ContactInfo
<template>
  <div class="container">
    <div class="row">
      Email: {{ emailAddress }}
      Twitter: {{ twitterHandle }}
      Instagram: {{ instagram }}
    </div>
  </div>
</template>
```

```
export default {
  name: 'ContactInfo',
  props: ['emailAddress', 'twitterHandle', 'instagram'],
};
```

The `ContactInfo` component takes the props `emailAddress`, `twitterHandle`, and `instagram`, and displays them on the page.

Our profile page component, `ProfilePage`, looks like this:

```
// ProfilePage
<template>
  <div class="profile-page">
    <div class="avatar">
      <img src="user.profilePicture" />
      {{ user.name }}
    </div>
  </div>
</template>
```

```
export default {
  name: 'ProfilePage',
  data() {
    return {
      // In a real app we would get this data from a server
      user: {
        name: 'John Smith',
        profilePicture: './profile-pic.jpg',
        emailAddress: 'john@smith.com',
        twitterHandle: 'johnsmith',
        instagram: 'johnsmith345',
      },
    }
  }
};
```

Our `ProfilePage` component currently displays the users profile picture along with their name. It also has the user data object.

How do we get that data from the parent component (`ProfilePage`) down into our child component (`ContactInfo`)?

We have to pass down this data using props.

First we need to import our `ContactInfo` component into the `ProfilePage` component:

```
// Import the component
import ContactInfo from './ContactInfo.vue';

export default {
  name: 'ProfilePage',

  // Add it as a dependency
  components: {
    ContactInfo,
  },

  data() {
    return {
      user: {
        name: 'John Smith',
        profilePicture: './profile-pic.jpg',
        emailAddress: 'john@smith.com',
        twitterHandle: 'johnsmith',
        instagram: 'johnsmith345',
      },
    }
  }
};
```

Second, we have to add in the component to our `<template>` section:

```
// ProfilePage
<template>
  <div class="profile-page">
    <div class="avatar">
      <img src="user.profilePicture" />
      {{ user.name }}
    </div>

    <!-- Add component in with props -->
    <contact-info
      :email-address="emailAddress"
      :twitter-handle="twitterHandle"
      :instagram="instagram"
    />

  </div>
</template>
```

Now all the user data that `ContactInfo` needs will flow down the component tree and into `ContactInfo` from the `ProfilePage`!

The reason we keep the data in `ProfilePage` and not `ContactInfo` is that other parts of the profile page need access to the user object.

Since **data only flows down**, this means we have to put our data high enough in the component tree so that it can flow down to all of the places it needs to go.

If you enjoyed this article or have any comments, let me know by replying to this tweet!