Vue Components

In Vue.js, components are the building blocks of a user interface. They are like custom elements that encapsulate a specific part of the UI along with its behavior and can be reused throughout your application.

Think of components as Lego bricks. Each brick is a self-contained piece that serves a specific purpose, and you can combine them to create more complex structures. Similarly, Vue components encapsulate a piece of your user interface and can be combined to form larger and more sophisticated UI elements.

Creating a Vue Component

Global Component (Vue.component)

To create a global component, you use the `Vue.component` method. This method allows you to register a component globally, meaning it can be used anywhere in your application.

```
<template>
  <div>
    <h1>{{ message }}</h1>
  </div>
</template>

<script>
// Register a global Vue component
Vue.component('my-component', {
  data() {
```

```
    return {
      message: 'Hello from My Component!'
    };
  }
});
</script>
```

In this example, we've defined a simple component called 'my-component'. It has a template with a heading element (`<h1>`) that displays the message "Hello from My Component!".

Single File Component (SFC)

While global components are useful for simple applications, as your project grows, it's better to use Single File Components (SFC). SFCs keep the template, script, and styles of a component in a single file.

```
<!-- MyComponent.vue -->
<template>
  <div>
    <h1>{{ message }}</h1>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello from My Component!'
    };
  }
};
</script>

<style>
/* Add styles specific to this component */
h1 {
  color: blue;
}
</style>
```

To use this SFC in your application, you'll import it and then register it locally in your parent component.

```
<template>
  <div>
    <my-component></my-component>
  </div>
</template>
```

```
<script>
import MyComponent from './MyComponent.vue';

export default {
  components: {
    MyComponent
  }
};
</script>
```

Now, you can use the `<my-component></my-component>` tag within the template of the parent component to include the 'MyComponent' inside it.

Component Communication

In Vue.js, there are three common ways to facilitate communication between components:

Props

Props are a way to pass data from a parent component to a child component in Vue.js. They allow you to share data down the component tree, enabling communication between parent and child components. Props are read-only in the child component, meaning the child component cannot directly modify the data received through props. This ensures a unidirectional flow of data, where changes to props in the parent component are automatically reflected in the child component.

Defining Props in the Child Component

To define props in the child component, you need to specify them in the `props` option of the component. Props are defined as an array of strings, where each string represents the name of the prop that you expect to receive from the parent component.

```
<!-- ChildComponent.vue -->
<template>
  <div>
    <p>{{ message }}</p>
  </div>
</template>

<script>
export default {
  props: ['message']
};
</script>
```

In this example, we have a simple `ChildComponent` that expects a prop called `message`.

Passing Props from the Parent Component

In the parent component, you can bind the value of a data property to the prop of the child component using the `v-bind` directive or its shorthand `:`. This way, the data from the parent component is passed to the child component as a prop.

```
<!-- ParentComponent.vue -->
<template>
  <div>
    <child-component :message="parentMessage"></child-component>
  </div>
</template>

<script>
import ChildComponent from './ChildComponent.vue';

export default {
  components: {
    ChildComponent
  },
  data() {
    return {
      parentMessage: 'Hello from Parent!'
    };
  }
};
</script>
```

In this example, the parent component (`ParentComponent`) passes the value of the `parentMessage` data property to the `message` prop of the child component (`ChildComponent`) using `v-bind` or the shorthand `:`.

Using Props in the Child Component

Once the data is passed from the parent to the child component as a prop, you can access and use it in the child component's template or script using the prop name, just like you would with any other data property.

```
<!-- ChildComponent.vue -->
<template>
  <div>
    <p>{{ message }}</p>
  </div>
</template>
```

```
<script>
export default {
  props: ['message']
};
</script>
```

In this example, the `message` prop received from the parent is displayed inside the `<p>` element in the child component's template.

Passing Dynamic Props

Props are reactive, so when the parent component's data changes, the child component's props are automatically updated. This allows for dynamic updates and reactivity between parent and child components.

```
<!-- ParentComponent.vue -->
<template>
  <div>
    <child-component :message="dynamicMessage"></child-component>
  </div>
</template>

<script>
import ChildComponent from './ChildComponent.vue';

export default {
  components: {
    ChildComponent
  },
  data() {
    return {
      dynamicMessage: 'Hello from Parent!'
    };
  },
  mounted() {
    // Simulating dynamic update after 2 seconds
    setTimeout(() => {
      this.dynamicMessage = 'Updated Message from Parent!';
    }, 2000);
  }
};
</script>
```

In this example, the `ParentComponent` updates the `dynamicMessage` data property after a delay. As a result, the child component (`ChildComponent`) will automatically reflect the updated prop value, and you'll see the message change in the child component's template.

Custom Events

Custom events in Vue.js allow child components to communicate with their parent components. This communication mechanism enables child components to emit events and notify their parent components about specific actions or changes that occurred within the child component.

Emitting Custom Events from Child Component:

To emit a custom event from a child component, you use the `$emit` method provided by Vue.js. The `$emit` method allows you to trigger a custom event along with optional data that can be passed to the parent component.

```
<!-- ChildComponent.vue -->
<template>
  <div>
    <button @click="sendMessageToParent">Send Message to Parent</button>
  </div>
</template>

<script>
export default {
  methods: {
    sendMessageToParent() {
      const message = 'Hello from Child!';
      this.$emit('custom-event', message);
    }
  }
};
</script>
```

In this example, the `ChildComponent` emits a custom event named `'custom-event'` when the button is clicked. It sends the message `'Hello from Child!'` along with the event.

Listening for Custom Events in Parent Component

To listen for custom events from a child component, you use the `v-on` directive (or its shorthand `@`) in the parent component's template. The `v-on` directive allows you to specify the custom event to listen for and the method that should be executed when the event occurs.

```
<!-- ParentComponent.vue -->
<template>
  <div>
    <child-component @custom-event="handleCustomEvent"></child-component>
    <p>Message from child: {{ messageFromChild }}</p>
  </div>
</template>
```

```
<script>
import ChildComponent from './ChildComponent.vue';

export default {
  components: {
    ChildComponent
  },
  data() {
    return {
      messageFromChild: ''
    };
  },
  methods: {
    handleCustomEvent(message) {
      this.messageFromChild = message;
    }
  }
};
</script>
```

In this example, the `ParentComponent` listens for the `'custom-event'` emitted by the `ChildComponent`. When the event is triggered, the `handleCustomEvent` method is executed, and it receives the message sent by the child component. The method then updates the `messageFromChild` data property, which is displayed in the template.

Passing Data with Custom Events

Custom events allow you to pass data from the child component to the parent component. This enables child components to provide valuable information to their parents.

```
<!-- ChildComponent.vue -->
<template>
  <div>
    <button @click="sendDataToParent">Send Data to Parent</button>
  </div>
</template>

<script>
export default {
  methods: {
    sendDataToParent() {
      const dataToSend = { name: 'John', age: 30 };
      this.$emit('custom-event', dataToSend);
    }
  }
};
</script>
```

In this updated example, the `ChildComponent` emits the `'custom-event'` with an object containing name and age data when the button is clicked.

Listening for Custom Events with Data in Parent Component

In the `ParentComponent`, you can access the data sent by the child component through the custom event. The data is accessible as the second argument in the method that handles the event.

```
<!-- ParentComponent.vue -->
<template>
  <div>
    <child-component @custom-event="handleCustomEvent"></child-component>
    <p>Name: {{ person.name }}, Age: {{ person.age }}</p>
  </div>
</template>

<script>
import ChildComponent from './ChildComponent.vue';

export default {
  components: {
    ChildComponent
  },
  data() {
    return {
      person: {
        name: '',
        age: ''
      }
    };
  },
  methods: {
    handleCustomEvent(data) {
      this.person = data;
    }
  }
};
</script>
```

In this updated example, the `ParentComponent` listens for the `'custom-event'` emitted by the `ChildComponent`. The `handleCustomEvent` method is executed when the event occurs, and it updates the `person` data property with the data sent by the child component.

Event Bus

The event bus is an alternative way to allow unrelated components to communicate with each other in a decoupled manner.

It acts as a central communication channel, where components can emit and listen for events without having direct parent-child relationships.

Creating the Event Bus

To set up the Event Bus, you typically create an instance of Vue that will act as the bus. This instance will be used solely for managing events and communication between components.

```
// event-bus.js
import Vue from 'vue';
export const eventBus = new Vue();
```

In this example, we create an instance called `eventBus`, which will be used as our central communication system.

Emitting Events (Sending Data)

Any component that wants to send data to other components emits an event on the event bus. The component emits the event with a specific name and optional data to be passed to other components.

```
// SenderComponent.vue
import { eventBus } from './event-bus';

export default {
  methods: {
    sendDataToOtherComponent() {
      const dataToSend = 'Hello from Sender!';
      eventBus.$emit('custom-event-name', dataToSend);
    }
  }
};
```

In this example, the `SenderComponent` emits a custom event named `'custom-event-name'` on the `eventBus`, along with the message `'Hello from Sender!'`.

Listening for Events (Receiving Data)

Any component that wants to receive data from other components listens for events on the event bus. When the specified event occurs, the component's callback function is executed, and the data sent along with the event can be accessed.

```
// ReceiverComponent.vue
import { eventBus } from './event-bus';

export default {
  data() {
    return {
      receivedData: ''
    };
  },
  created() {
    eventBus.$on('custom-event-name', (dataReceived) => {
      this.receivedData = dataReceived;
    });
  }
};
```

In this example, the `ReceiverComponent` listens for the `'custom-event-name'` event on the `eventBus`. When the event is emitted by the `SenderComponent`, the callback function is triggered, and the received data (in this case, `'Hello from Sender!'`) is stored in the `receivedData` property of the `ReceiverComponent`.

Cleanup (Optional)

When a component is destroyed (e.g., when navigating away from a page or unmounting), it's a good practice to clean up and remove any event listeners associated with the component. Otherwise, you might have unnecessary listeners still active in the memory.

```
// ReceiverComponent.vue
import { eventBus } from './event-bus';

export default {
  // ...
  beforeDestroy() {
    eventBus.$off('custom-event-name'); // Remove the event listener
  }
};
```

In this example, we remove the event listener for the `'custom-event-name'` event in the `beforeDestroy` lifecycle hook to avoid any potential memory leaks.

Using the Event Bus, you can create a loosely coupled architecture in your Vue.js application, where components can communicate with each other without direct dependencies. However, keep in mind that while the Event Bus can simplify communication between components, excessive usage of it might make the data flow

harder to follow and maintain. It's essential to use it judiciously and consider other communication patterns like props and custom events, depending on the specific needs of your application.

Parent-Child Component Relationship and Data Flow

In Vue.js, components can be arranged in a hierarchical structure, forming a parent-child relationship. Data flows down from the parent component to the child component using props. Custom events allow the child component to communicate back to the parent component.

```
+-----------+
| Parent    |
| Component |
+-----------+
    ↑
    | (Data flows down using props)
    ↓
+-----------+
| Child     |
| Component |
+-----------+
    ↑
    | (Communication via custom events)
    ↓
+-----------+
| Grandchild|
| Component |
+-----------+
```

Custom events in Vue.js allow child components to communicate with their parent components. This communication mechanism enables child components to emit events and notify their parent components about specific actions or changes that occurred within the child component.

Component Lifecycle Hooks

Component lifecycle hooks in Vue.js allow you to perform actions at different stages of a component's existence. Understanding the component lifecycle is crucial for managing data, performing initialization, making API calls, and cleaning up resources during the component's life.

created

The `created` hook is called when the component is created but before it is mounted to the DOM.

It is an excellent place to perform initial setup, such as setting up data, creating event listeners, or making API calls. At this stage, the template has been compiled, but it has not yet been rendered in the DOM.

Use the `created` hook to initialize data, set up event listeners, or perform API calls that are required for the component to function correctly.

- Fetch initial data from an API and populate the component's data properties.
- Attach event listeners to handle user interactions or listen for events from external sources.
- Perform any other setup tasks that do not require access to the DOM.

mounted

The `mounted` hook is called after the component has been mounted to the DOM. It is a suitable place for interacting with the DOM or performing tasks that require access to the DOM elements. At this stage, the component's template has been rendered in the DOM, and you can manipulate the DOM or perform side effects.

- Use the `mounted` hook for tasks that require access to the DOM, such as interacting with the component's rendered elements or setting up third-party libraries.
- Manipulate the DOM, e.g., setting focus on an input element, initializing a map, or initializing a chart.
- Register event listeners directly on DOM elements, especially for non-Vue events (e.g., native DOM events).

updated

The `updated` hook is called after the component has been re-rendered due to changes in its reactive data or props. It is useful for performing tasks that need to react to changes in the component's data or update the DOM in response to data changes. However, be cautious about triggering state changes within this hook, as it can lead to an infinite loop.

- Use the `updated` hook when you need to react to changes in the component's data or the DOM after a re-render.

- Perform additional actions after the component updates, such as updating the DOM or making side-effect API calls based on new data or prop changes.
- Be cautious about triggering state changes within this hook to avoid infinite loops.

beforeDestroy

The `beforeDestroy` hook is called just before a component is about to be destroyed.

It is a good place to perform cleanup tasks, such as removing event listeners or cleaning up resources to prevent memory leaks. The component is still fully functional at this point, and it has access to its data and methods.

destroyed

The `destroyed` hook is called after the component has been destroyed, and it is no longer active. At this stage, all data, methods, and DOM elements related to the component have been removed. It is the last opportunity to perform any cleanup or additional actions related to the component.

- Use the `destroyed` hook when you need to perform additional actions or cleanup after the component has been destroyed.
- Typically, you do not need to use this hook frequently, as the component is no longer functional at this point.
- This hook can be helpful if you have complex cleanup tasks that should run only after the component is completely removed from the DOM.

Here's a visual representation of the component lifecycle:

```
created --> mounted --> (Data or Props Changes) -->
updated --> (Component is about to be destroyed) -->
beforeDestroy --> destroyed
```

It's important to note that these hooks are not meant to handle data flow or state management directly. For that purpose, you should use data properties, props, and Vuex (for larger applications) to manage the state of your application. Lifecycle hooks are mainly intended for performing setup and cleanup tasks related to the component.

Example of using lifecycle hooks in different scenarios:

```
<template>
  <div>
```

```
    <p>{{ message }}</p>
    <input v-model="inputValue" />
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello from Vue!',
      inputValue: ''
    };
  },
  created() {
    // Fetch data from an API and populate component data
    this.fetchDataFromAPI();
  },
  mounted() {
    // Set focus on the input element after the component is mounted
    this.$nextTick(() => {
      this.$refs.input.focus();
    });
  },
  updated() {
    // Perform an action when the input value changes
    console.log('Input value updated:', this.inputValue);
  },
  beforeDestroy() {
    // Clean up resources and unregister event listeners before the component is destroyed
    this.cleanupTasks();
  },
  destroyed() {
    // Perform additional cleanup or actions after the component is destroyed
    console.log('Component destroyed');
  },
  methods: {
    fetchDataFromAPI() {
      // Perform an API call and update component data
    },
    cleanupTasks() {
      // Clean up resources, event listeners, etc.
    }
  }
};
</script>
```

In this example, we have a component with various lifecycle hooks used for different purposes:

- `created`: We use it to fetch initial data from an API and populate the component's data.
- `mounted`: We set focus on the input element after the component is mounted to the DOM.

- `updated`: We log the input value whenever it changes.
- `beforeDestroy`: We perform cleanup tasks, such as unregistering event listeners, before the component is destroyed.
- `destroyed`: We log a message to indicate that the component has been destroyed.