Basic Vue.js Concepts

Create a New Project

Open your terminal or command prompt and run the following command to create a new Vue.js project named "my-vue-project":

```
vue create my-vue-project
```

The Vue CLI will prompt you to pick a preset. You can either choose the default preset or manually select features. Once the project is created, navigate into the project directory:

`cd my-vue-project`

Project Structure

The Vue CLI generates the following file structure for your project:

```
my-vue-project/
├── public/
│ ├── index.html
│ └── ...
├── src/
│ ├── assets/
│ │ └── ...
│ ├── components/
│ │ └── ...
│ ├── App.vue
│ └── main.js
├── package.json
├── vue.config.js
└── ...
```

- `public/`: This directory contains the publicly accessible files for your project, such as the `index.html` file where your Vue app will be mounted.
- `src/`: This is the main source directory where your Vue application code resides.
- `assets/`: Place your static assets like images, fonts, etc., inside this directory.
- `components/`: This is where you'll organize your Vue components, dividing them into separate files for better maintainability.
- `App.vue`: The main Vue component that acts as the root of your application. It's a combination of template, script, and style all in one file.
- `main.js`: This is the entry point of your application, where you create the Vue instance and configure it.
- `package.json`: The file that stores your project's metadata and dependencies.
- `vue.config.js`: A configuration file that allows you to customize Vue CLI's behavior.

Vue Instance in main.js

Open the `main.js` file located inside the `src/` directory. This is where you'll create the Vue instance:

```
import Vue from 'vue';
import App from './App.vue';
```

```
new Vue({ render: h => h(App)
}).$mount('#app');
```

We import Vue and the `App.vue` component.

Inside the Vue instance, we use the `render` function to render the `App` component.

The `.$mount('#app')` method binds the Vue instance to the HTML element with the id "app" in the `index.html` file.

Using the Vue Instance:
The `App.vue` component is where you can start using the Vue instance and its features. This file consists of a `<template>`, `<script>`, and `<style>` section.

Here's a simplified example of `App.vue`:

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

<script>
export default {
  data() {
    return {
      message: 'Hello, Vue!'
    };
  }
};
</script>

<style>
/* Your component styles go here */
</style>
```

In this example, we have a Vue component that displays the message "Hello, Vue!" using data binding (`{{ message }}`). The data is defined in the `data()` method of the component, which is reactive, meaning any changes to `message` will automatically update the template.

Data binding

There are two main types of data binding in Vue.js:

Text Interpolation (Double Curly Braces)

Text interpolation is used to display dynamic data in your HTML template. It involves using double curly braces `{{ }}` to insert Vue data properties directly into the template.

Let's consider an example of displaying a user's name:

```
<template>
  <div>
    <p>Welcome, {{ username }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      username: 'John Doe'
    };
  }
};
</script>
```

In this example, the `username` data property in the Vue instance is bound to the template using text interpolation. The `{{ username }}` expression will be replaced with the value of the `username` data property, resulting in the output: "Welcome, John Doe".

As you modify the `username` data property programmatically in your Vue instance, the displayed text will automatically update to reflect the new value.

Attribute and Property Binding (v-bind):

Attribute and property binding are used to bind dynamic values to HTML attributes and properties. You use the `v-bind` directive or its shorthand `:` to achieve this.

Let's consider an example of dynamically setting the `src` attribute of an image:

```
<template>
  <div>
    <img v-bind:src="imageURL" alt="Dynamic Image">
  </div>
</template>

<script>
export default {
  data() {
    return {
      imageURL: 'https://example.com/image.jpg'
    };
```

```
  }
};
</script>
```

In this example, we use `v-bind:src` or `:src` to bind the value of the `imageURL` data property to the `src` attribute of the `img` element. The `imageURL` data property holds the URL of the image that will be displayed.

When the Vue instance is mounted, the `src` attribute of the `img` element will be set to the value of `imageURL`, and the image will be displayed accordingly.

Attribute and property binding also allows you to set values for other attributes and properties like `href`, `class`, `disabled`, and more.

Vue directives

Vue directives are special attributes that you can add to HTML elements to enhance their functionality and behavior. Directives start with the `v-` prefix, and they allow you to perform various tasks, such as conditional rendering, rendering lists, event handling, and attribute binding.

v-if and v-else

The v-if directive is used for conditional rendering. It evaluates an expression and only renders the element if the expression is truthy. If the expression is falsy, the element is not rendered. You
can also use v-else to define an alternative element that will be rendered when the v-if condition is not met.

```
<template>
  <div>
    <p v-if="isLoggedIn">Welcome, {{ username }}!</p>
    <p v-else>Please log in to continue.</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      isLoggedIn: true,
      username: 'John Doe'
    };
  }
};
</script>
```

v-for

The v-for directive is used for rendering lists based on an array in your Vue instance. It iterates through each item in the array and creates a copy of the element for each item.

```
<template>
  <ul>
    <li v-for="item in items" :key="item.id">{{ item.name }}</li>
  </ul>
</template>

<script>
export default {
  data() {
    return {
      items: [
        { id: 1, name: 'Item 1' },
        { id: 2, name: 'Item 2' },
        { id: 3, name: 'Item 3' }
      ]
    };
  }
};
</script>
```

v-on (or @)

The v-on directive (or its shorthand @) is used for event handling. It allows you to attach event listeners to HTML elements and trigger methods or expressions when the events occur.

```
<template>
  <button @click="handleClick">Click Me</button>
</template>

<script>
export default {
  methods: {
    handleClick() {
      alert('Button clicked!');
    }
  }
};
</script>
```

v-bind (or :)

The v-bind directive (or its shorthand:) is used for attribute binding. It allows you to dynamically bind values to HTML element attributes or properties.

```
<template>
  <a :href="linkURL">Click here</a>
</template>

<script>
export default {
  data() {
    return {
      linkURL: 'https://example.com'
    };
  }
};
</script>
```

v-show

Similar to `v-if`, the `v-show` directive also controls the visibility of elements based on an expression. However, unlike `v-if`, v-show does not remove the element from the DOM; instead, it toggles the `display` CSS property to show or hide the element.

```
<template>
  <div>
    <p v-show="isVisible">This element is shown or hidden</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      isVisible: true
    };
  }
};
</script>
```

v-cloak

The `v-cloak` directive is used to hide uncompiled Vue.js templates until the Vue instance has finished compiling. This directive is typically used to prevent the display of unstyled or "flashing"
content before Vue is fully loaded.

```
<template>
  <div v-cloak>
    <!-- Your Vue.js template content -->
  </div>
</template>
```

v-model

The `v-model` directive provides two-way data binding for form elements. It automatically synchronizes the data between the form input and the Vue instance, making it easy to handle user input.

```
<template>
  <div>
    <input type="text" v-model="inputValue">
    <p>You entered: {{ inputValue }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      inputValue: ''
    };
  }
};
</script>
```

v-text

The `v-text` directive is used to set the text content of an element directly. It behaves similarly to the double curly braces `{{ }}` syntax for text interpolation, but with `v-text`, you can't use expressions or HTML tags.

```
<template>
  <div>
    <p v-text="message"></p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello, Vue!'
    };
  }
```

```
};
</script>
```

v-html

The `v-html` directive allows you to render HTML content from a data property. Be cautious when using `v-html` as it may expose your application to potential security risks if the content comes from an untrusted source.

```
<template>
  <div>
    <p v-html="htmlContent"></p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      htmlContent: '<strong>This is rendered as HTML</strong>'
    };
  }
};
</script>
```

Basic Event Handling

In Vue.js, you can handle events using the `v-on` directive, also known as `@` (shorthand). This directive allows you to attach event listeners to HTML elements and execute methods or expressions when the events are triggered.

To handle an event, add the `v-on` directive followed by the event name to the HTML element you want to listen to. Then, assign the method or expression you want to execute when the event occurs.

```
<template>
  <div>
    <button v-on:click="handleClick">Click Me</button>
  </div>
</template>

<script>
export default {
  methods: {
    handleClick() {
      alert('Button clicked!');
    }
  }
```

```
};
</script>
```

In this example, the `handleClick` method is executed when the button is clicked. The `v-on:click` directive listens for the "click" event on the button element and triggers the `handleClick` method.

Event Modifiers

Event modifiers are special postfixes that you can add to the event handling expression to modify the behavior of the event. For example:

- `.stop`: Stops the event propagation. It prevents the event from triggering the same event on the parent elements.
- `.prevent`: Prevents the default behavior of the event. For example, it prevents a form submission or a link navigation.
- `.capture`: Adds an event listener in the capture phase rather than the default bubbling phase.
- `.self`: Only triggers the event handler if the event target is the element itself, not its children.
- `.once`: Attaches the event handler to trigger only once.

```
<template>
  <div>
    <!-- The event propagation is stopped -->
    <button v-on:click.stop="handleClick">Click Me</button>

    <!-- The default form submission is prevented -->
    <form v-on:submit.prevent="submitForm">
      <input type="text">
      <button type="submit">Submit</button>
    </form>

    <!-- The event handler will only trigger once -->
    <button v-on:click.once="handleClickOnce">Click Once</button>
  </div>
</template>

<script>
export default {
  methods: {
    handleClick() {
      alert('Button clicked!');
    },
    submitForm() {
      // Handle form submission
```

```
    },
    handleClickOnce() {
      alert('This will trigger only once.');
    }
  }
};
</script>
```

Passing Event Data:

You can also pass additional data to the event handler using the special `$event` variable. This variable holds the event object, which provides information about the event.

```
<template>
  <div>
    <button v-on:click="handleClick($event, 'Additional Data')">Click Me</button>
  </div>
</template>

<script>
export default {
  methods: {
    handleClick(event, additionalData) {
      alert('Button clicked!');
      console.log('Additional Data:', additionalData);
    }
  }
};
</script>
```

In this example, the `handleClick` method receives the event object as the first argument and the additional data as the second argument.

Computed Properties

Computed properties are a powerful feature in Vue.js that allow you to calculate and derive new data properties based on existing ones. They are used to perform complex calculations or data transformations in a way that is more efficient and maintainable compared to using methods.

Defining a Computed Property

Computed properties are defined in the `computed` option of a Vue component. To create a computed property, you define a function that returns the computed value based on other data properties.

```
<template>
  <div>
    <p>Radius: {{ radius }}</p>
    <p>Area: {{ area }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      radius: 5
    };
  },
  computed: {
    area() {
      return Math.PI * this.radius * this.radius;
    }
  }
};
</script>
```

In this example, the computed property `area` calculates the area of a circle based on the `radius` data property. When `radius` changes, the `area` property will automatically update to reflect the new calculated value.

Difference Between Computed Properties and Methods

While computed properties and methods can both calculate and derive data, there are some key differences:

- **Caching:** Computed properties are cached based on their dependencies. Vue will only recompute the value when one of its dependencies changes. This means that if a computed property's dependencies have not changed, Vue will return the cached value instead of recomputing it. In contrast, methods are always recalculated when they are called.
- **Reactivity:** Computed properties are reactive, just like data properties. This means they automatically update when any of their dependencies change. Methods, on the other hand, don't have this reactivity behavior. If you want a method to update the template whenever it is called, you would need to explicitly call it in the template using `{{ }}`.
- **Syntax:** Computed properties are accessed like data properties in the template, without using parentheses. Methods are called with parentheses in the template.

When to Use Computed Properties and Methods:
Use computed properties when you need to calculate and display a value based on other data properties and when you want to take advantage of caching and automatic reactivity. Computed properties are especially useful when you have complex calculations or transformations that should update automatically when their dependencies change.

Use methods when you need to perform an action or execute code that doesn't depend on any reactive data. Methods are helpful for handling user interactions, performing operations not related to data, or returning dynamic data that requires additional processing each time it's called.

In general, if a value is dependent on other data and should be automatically updated, use a computed property. If it's an action or dynamic value not related to data, use a method.

Remember that using computed properties can help improve the performance of your application, as Vue.js intelligently caches and only updates the computed properties that are affected by data changes. As a best practice, reserve methods for actions and computed properties for derived values.

Watchers

Watchers are a Vue.js feature that allows you to react to changes in data properties and perform custom logic in response. Unlike computed properties, which are used to derive new data based on existing data, watchers are better suited for handling asynchronous or expensive operations, such as making API calls or performing time-consuming computations.

Defining a Watcher

Watchers are defined in the `watch` option of a Vue component. To create a watcher, you specify the name of the data property you want to watch, and then provide a handler function that will be executed when the property changes.

```
<template>
  <div>
    <p>Current Value: {{ value }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
```

```
      value: 0
    };
  },
  watch: {
    value(newValue, oldValue) {
      // Watcher handler function
      console.log('New Value:', newValue);
      console.log('Old Value:', oldValue);
    }
  }
};
</script>
```

In this example, we have a data property `value`, and we define a watcher for it. The watcher will execute the handler function whenever `value` changes. The handler function receives two arguments: the new value (`newValue`) and the old value (`oldValue`) of the data property.

Async Operations with Watchers:

Watchers are particularly useful when you need to perform asynchronous operations in response to data changes. For example, you might want to make an API call when a particular data property changes and update other data properties based on the API response.

```
<template>
  <div>
    <p>Current Value: {{ value }}</p>
    <p>Result: {{ result }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      value: 0,
      result: null
    };
  },
  watch: {
    value: {
      immediate: true,
      handler(newValue) {
        // Watcher handler function with async operation
        this.getResultFromAPI(newValue);
      }
    }
  },
  methods: {
    async getResultFromAPI(value) {
```

```
      try {
        // Simulate an asynchronous API call with a timeout
        const response = await new Promise((resolve) => {
          setTimeout(() => resolve({ data: value * 2 }), 1000);
        });

        this.result = response.data;
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    }
  }
};
</script>
```

In this example, we use a watcher to execute the `getResultFromAPI` method whenever the `value` data property changes. The method makes an asynchronous API call using `setTimeout` to simulate the delay. When the API call resolves, the `result` data property is updated with the response.

Watcher Options

Watchers have several options that can be used to fine-tune their behavior:

`handler`: The function to execute when the watched property changes.

`immediate`: If set to `true`, the watcher will be triggered immediately when the component is created, even if the data property's value has not changed yet.

`deep`: If set to `true`, the watcher will perform a deep watch, meaning it will also watch for changes in nested properties of the data property.

`deep` and `immediate` can be used together to execute the watcher immediately and also trigger it on any changes in the nested properties.