Accelerating Application Design With OpenAPI

Getting The Tools

Before you begin, you will need to ensure that your development environment is properly configured. To make that happen you will want certain tools installed and usable. Most of this tutorial will assume a UNIX-like environment (MacOS, Linux, WSL2).

- Install NodeJS
- Install a Java Virtual Machine
- Install Yarn

Bootstrap Your Project

- Install Vue CLI
  - `yarn global add @vue/cli@latest`
- Create your new project. We will be creating a household bills manager and cash flow analyzer
  - `vue create todo-quasar-openapi`
- Change to the new project directory
  - `cd todo-quasar-openapi`
- Add Quasar
  - `vue add quasar`
- Add Dev dependencies
  - `yarn add -D @openapitools/openapi-generator-cli @stoplight/prism npm-watch`
- Create a basic OpenAPI contract

```
openapi: 3.0.2
info:
  title: Todo
  version: 1.0.0
  description: My Application
servers:
  - url: "http://{domain}:{port}{base_path}"
    description: "API URL"
    variables:
      base_path:
        enum:
          - /
          - /api/v1
```

```
        default: /
      domain:
        enum:
          - localhost
        default: localhost
      port:
        enum:
          - '7080'
        default: '7080'
tags:
  - name: api
paths:
  /health:
    get:
      operationId: getHealth
      responses:
        '200':
          description: 'OK'
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Errors'
        default:
          $ref: '#/components/responses/Error'
components:
  responses:
    Error:
      description: Error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Errors'
  schemas:
    Errors:
      type: object
      required:
        - code
      properties:
        timestamp:
          type: string
          format: date-time
        msg:
          type: string
        code:
          type: number
          format: int64
```

This is a reasonable start for any new REST API contract

- Add new scripts to `package.json`

```
"watch": "npm-watch",
"prism": "prism mock -d --cors -p 7080 openapi.yml",
"openapi": "rm -f src/sdk; mkdir src/sdk; openapi-generator-cli generate -g
typescript-axios -i openapi.yml -o src/sdk/ -p
withSeparateModelsAndApi=true,apiPackage=api,modelPackage=models"
```

- Add `watch` block to `package.json` after the `scripts` block

```
"watch": {
  "openapi": {
    "patterns": [     ①
      "openapi.yml",
      "package.json"
    ],
    "inherit": true
  },
  "prism": true,      ②
  "serve": {
    "patterns": [     ③
      "yarn.lock",
      "package.json",
      "src/main.ts",
      "src/quasar-user-options.ts",
      "tsconfig.json",
      "vue.config.js",
      "babel.config.js"
    ],
    "inherit": true
  }
},
```

① When either `openapi.yml` or `package.json` change, regenerate the OpenAPI Client code

② Ensure that the `prism` mock API server is running. It will automatically detect changes in the OpenAPI file.

③ When any of the core framework files change, restart the development web server

*What did I just accomplish?

You have just created a new project using the Quasar framework for VueJS. You also added tooling which will allow you to both create a Mock API server (using Prism) but also generate the code which allows you to talk to that API automatically. As we proceed, you will see that when we need a new data type or new API method, we can quickly add it to the `openapi.yml` file and the `npm-watch` tool will automatically regenerate the necessary code and restart the necessary services.

Open your project in your preferred IDE

These are IDE's I have had good luck with

- VSCode
- WebStorm

Clean Up Some Template Issues

- Rename `src/quasar-user-options.js` to `src/quasar-user-options.ts` to eliminate TypeScript validation errors
- Replace `<HelloWorld />` with `<router-view />` in `src/App.vue` and remove all other references to `HelloWorld` in that file. (It is handled in a different component now)
- Add a type interface to `App.vue` and use it for the `setup()` function:

```
<template>
  <q-layout view="hHh lpR fFf">
    <q-header elevated style="height: 5vh;">
      <q-toolbar>
        <q-toolbar-title>
          Todo List
        </q-toolbar-title>
      </q-toolbar>
    </q-header>
    <q-page-container style="height: 100%;">
      <router-view />
    </q-page-container>
  </q-layout>
</template>
<script lang="ts">
import { defineComponent, ref } from "vue";

export default defineComponent(() => {
  return {
    leftDrawerOpen: ref(false),
  }
});
</script>
```

Start Building Todo User Interface

- We know that we're going to need a Todo object type, so let's create that in the `openapi.yml`

```yaml
components:
  schemas:
    NewTodo:
      type: object
      required:
      - title
      properties:
        title:
          type: string
          maxLength: 255
        description:
          type: string
        id:
          type: string
          format: uuid
```

- That will be a good object definition for when we are creating a new Todo item, but we also want some validation, so let's create a `Todo` type which has some required fields:

```yaml
components:
  schemas:
    Todo:
      type: object
      required:
      - title
      - id
      allOf:
      - $ref: '#/components/schemas/NewTodo'
```

- Let's add a new endpoint to let us get the complete list of Todos

```
tags:
  - name: api
  - name: todo    ①
paths:
  /todos:
    get:
      description: Get all todos
      operationId: getAllTodos    ②
      tags:
        - todo
      responses:
        '200':
          description: 'OK'
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Todo'
```

① The tag becomes the name of the API object for this tag

② The `operationId` becomes the method name in the API object to call in order to access that endpoint

- Once we add these, save the file and start our `watch` script

```
 yarn watch
yarn run v1.22.17
warning ../package.json: No license field
$ npm-watch
No task specified. Will go through all possible tasks
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): yarn.lock package.json src/main.ts src/quasar-user-
options.ts tsconfig.json vue.config.js babel.config.js
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `npm run -s serve`
[nodemon] 2.0.15
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): openapi.yml
[nodemon] watching extensions: js,mjs,json
[nodemon] watching path(s): openapi.yml package.json
[nodemon] starting `npm run -s prism`
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `npm run -s openapi`
```

```
rm: cannot remove 'src/sdk': Is a directory
mkdir: cannot create directory ‘src/sdk’: File exists
 INFO  Starting development server...
[3:43:15 PM] ▯ [CLI] ⋯  awaiting  Starting Prism⋯
[3:43:15 PM] ▯ [CLI] ▯  info      GET        http://127.0.0.1:7080/todos
[3:43:15 PM] ▯ [CLI] ▯  info      GET        http://127.0.0.1:7080/health
[3:43:15 PM] ▯ [CLI] ▯  start     Prism is listening on http://127.0.0.1:7080
[main] INFO  o.o.codegen.DefaultGenerator - Generating with dryRun=false
[main] INFO  o.o.codegen.DefaultGenerator - OpenAPI Generator: typescript-fetch
(client)
[main] INFO  o.o.codegen.DefaultGenerator - Generator 'typescript-fetch' is considered
stable.
[main] INFO  o.o.c.l.AbstractTypeScriptClientCodegen - Hint: Environment variable
'TS_POST_PROCESS_FILE' (optional) not defined. E.g. to format the source code, please
try 'export TS_POST_PROCESS_FILE="/usr/local/bin/prettier --write"' (Linux/Mac)
[main] INFO  o.o.c.l.AbstractTypeScriptClientCodegen - Note: To enable file post-
processing, 'enablePostProcessFile' must be set to `true` (--enable-post-process-file
for CLI).
[main] INFO  o.o.codegen.utils.ModelUtils - [deprecated] inheritance without use of
'discriminator.propertyName' has been deprecated in the 5.x release. Composed schema
name: null. Title: null
[main] INFO  o.o.codegen.utils.ModelUtils - [deprecated] inheritance without use of
'discriminator.propertyName' has been deprecated in the 5.x release. Composed schema
name: null. Title: null
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-
openapi/src/sdk/models/NewTodo.ts
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/models/Todo.ts
[main] INFO  o.o.codegen.utils.ModelUtils - [deprecated] inheritance without use of
'discriminator.propertyName' has been deprecated in the 5.x release. Composed schema
name: null. Title: null
[main] INFO  o.o.codegen.utils.ModelUtils - [deprecated] inheritance without use of
'discriminator.propertyName' has been deprecated in the 5.x release. Composed schema
name: null. Title: null
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-
openapi/src/sdk/apis/DefaultApi.ts
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/apis/TodoApi.ts
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/index.ts
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/runtime.ts
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/apis/index.ts
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/models/index.ts
[main] INFO  o.o.codegen.TemplateManager - Skipped
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/.openapi-
generator-ignore (Skipped by supportingFiles options supplied by user.)
```

```
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/.openapi-
generator/VERSION
[main] INFO  o.o.codegen.TemplateManager - writing file
/home/dphillips/Documents/RedHat/Workspace/todo-quasar-openapi/src/sdk/.openapi-
generator/FILES
################################################################################
# Thanks for using OpenAPI Generator.                                          #
# Please consider donation to help us maintain this project 🙏                 #
# https://opencollective.com/openapi_generator/donate                          #
################################################################################
[nodemon] clean exit - waiting for changes before restart
98% after emitting SizeLimitsPlugin

DONE  Compiled successfully in 5779ms
```

*What is happening here?

By defining the `NewTodo` and `Todo` schemas along with the `/todo` **GET** operation in the `openapi.yml` file and starting the watch, prism and openapi-generator start up the mock API server and generate the client-side code for talking to the API. The API client code can be found in `src/sdk` and we will use it to talk to the mock API as we develop the user interface application.

Create State Management For The Application With Pinia

Pinia is a state management extension for VueJS and it offers a way of reducing trashing and complexity by allowing you to centrally manage the state information in your web application. In this application, we will use it to make integration with our API simpler and more efficient.

- Create the subdirectory `src/stores` and create a new file there called `APIPlugin.ts`
  - This file will become an extension plugin for Pinia which allows us to manage API calls centrally
- In `APIPlugin.ts` we're going to embed our API client(s) into the state management system by extending the pinia context:

```
import { Configuration } from './../sdk/runtime';
import { PiniaPluginContext } from "pinia";

declare module 'pinia' {

    export interface PiniaCustomProperties<Id, S, G, A> {
        id: Id            ①
        state?: () => S
        getters?: G
        actions?: A
    }

    export interface DefineStoreOptionsInPlugin<Id extends string, S extends
StateTree, G, A> extends Omit<DefineStoreOptions<Id, S, G, A>, 'id' | 'actions'> {
        apiConfig: Configuration  ②
    }
}
```

① These are the default properties inside of a Pinia context

② We are extending the options for Pinia to allow it to store our API configuration object, the code for which was generated by `openapi-generator` via our `watch`

- Add the `TodoApi` to the Pinia custom properties

```
import { Configuration, TodoApi } from "@/sdk";
import { PiniaPluginContext } from "pinia";

declare module 'pinia' {

    export interface PiniaCustomProperties<Id, S, G, A> {
        id: Id
        state?: () => S
        getters?: G
        actions?: A
        todoApi: TodoApi    ①
    }

    export interface DefineStoreOptionsInPlugin<Id extends string, S extends
StateTree, G, A> extends Omit<DefineStoreOptions<Id, S, G, A>, 'id' | 'actions'> {
        apiConfig: Configuration
    }
}

export const APIPlugin = ({options, store}: PiniaPluginContext): void => { ②
    const { apiConfig } = options;

    if (apiConfig) {
        store.todoApi = new TodoApi(apiConfig)
    } else {
        store.todoApi = new TodoApi()
    }
}
```

① Add the new field definition so that we can attach the API object to the state properties

② Override the Pinia context so that it initializes the API client on load

- Initialize Pinia in the `src/main.ts` file of the Vue application

```
import { createPinia } from 'pinia'
import { APIPlugin } from './stores/APIPlugin';

const pinia = createPinia();
pinia.use((context) => APIPlugin(context));

createApp(App)
    .use(Quasar, quasarUserOptions)
    .use(router)
    .use(pinia)
    .mount('#app')
```

Create Our First Pinia Store

- Create a new file `src/stores/TodoStore.ts`
- In that file, define a `State` interface which will define what we keep in this store:

```
import { Todo } from '@/sdk';

interface State {
    todos: Todo[]
}
```

- Now, let's define our Todos store:

```
import { Todo } from '@/sdk';
import { defineStore } from 'pinia';

interface State {
    todos: Todo[]
}

export const initState = (): TodoState => ({    ①
    todos: [],
});

export const todoStore = defineStore('todos', {
    state: initState,

    getters: {
        todoList: (state) => state?.todos       ②
    },

    actions: {
        async loadTodos() {                     ③
            try {
                const { data } = await this.todoApi.getAllTodos();
                this.updateTodos(data);
            } catch (err) {
                // Do something with the error?
            }
        },
        updateTodos(todos: Todo[]) {            ④
            this.todos = todos;
        }
    }
});
```

① Initialize the State object with it's default values (an empty array)

② Create a getter which we can use in our components to retrieve data from the store

③ Create an action which uses our API client code to load Todos from the REST service

④ Create another method which applies the todos to the state once our Async method completes

- Let's use that newly created store in our `Home.vue` view

Use Our First API Call

- Open the `src/views/Home.vue` file and set the template as follows:

```
<template>
  <div class="flex q-pa-md" style="margin: 0;">
    <div :class="headerClasses">
      <div class="col-grow">
        Title/Description
        <q-btn icon="refresh" dense flat @click="reload" />
      </div>
    </div>
    <q-scroll-area style="height: 82vh; width: 100vw;">
      <q-list>
        <q-item v-for="todo in todoList" :key="todo.id" class="row datatable">
          <q-item-section class="col-grow">
            <q-expansion-item :label="todo.title">
            <template v-slot:header>
              <span class="title">{{ todo.title }}</span>
            </template>
            <template v-slot:default>
              <span class="description">{{ todo.description }}</span>
            </template>
            </q-expansion-item>
          </q-item-section>
        </q-item>
      </q-list>
    </q-scroll-area>
</template>
```

- Add the following code to the `<script>` block

```
import { todoStore } from "@/stores/TodoStore"
import { computed, defineComponent } from "vue";

export default defineComponent(() => {
  const todos = todoStore();                    ①

  todos.loadTodos();                            ②

  return {
    todoList: computed(() => todos.todoList)    ③
  }
})
```

① Instantiate our Pinia store

② Load the Todo items from the API service

③ Map the `todoList` from the store to a computed/reactive property

- We also need to add some items to the `<style>` block:

```
<style lang="sass" scoped>
.header
  font-size: 2.6vh
  font-weight: 800
  text-decoration: underline
  height: 3vh
  width: 100%

.scroll-area
  position: relative
  top: 3vh
  width: 100vw
  min-height: 87vh
  box-sizing: content-box

.datatable
  padding: 0px !important
  &:nth-child(odd)
    background-color: rgba(2,123,227,0.07)

.small-cell
  min-width: 2rem
  max-width: 2rem
  flex-direction: column
  justify-content: flex-start
  margin: 0px
  margin-top: 0.5rem
  padding: 0px

.title
  font-weight: 700
  font-size: 1.2rem

.description
  padding-left: 2rem
</style>
```

- Now, you should see the page reload and a table of Todo items! That's fantastic, but what if our API is accessed over a slow link or our servers are overloaded? How can we add a loading indicator?

Using A Loading Indicator

- First, let's add one of the Quasar `QAjaxBar` elements to our template in the `Home.vue` file. It can be places anywhere in the template.

```
    <q-ajax-bar
      ref="progressBar"
      position="bottom"
      color="red-8"
      size="0.75rem"
      skip-hijack
    />
```

The `ref` attribute allows us to use the `ref()` method to get a reference to this component in the script block

- In the script block, get a reference to the `QAjaxBar` and create some anonymous functions from it

```
export default defineComponent(() => {
  const progressBar = ref<QAjaxBar>();
  const incrementer = (increment?: number) => progressBar?.value?.increment(
increment);
  const stop = () => progressBar?.value?.stop();
```

- Create a wrapper around our `loadTodos` method from the `TodoStore`

```
  function loadTodos() {
    progressBar?.value?.start();
    todos.loadTodos(increment, stop);
  }

  loadTodos();
```

- Open up our `TodoStore.ts` store definition and modify the `loadTodos` action method:

```
async loadTodos(increment: (increment?: number) => void, stop: () => void) {
    increment(10);                                          ①
    const axiosConfig = {
        onUploadProgress: (progressEvent: ProgressEvent) => {
            increment(progressEvent.loaded * 80);           ②
        }
    }
    increment(20);                                          ③
    try {
        const { data } = await this.todoApi.getAllTodos(axiosConfig);
        this.updateTodos(data);
        stop();                                             ④
    } catch (err) {
        // Do something with the error?
    }
    increment(100);                                         ⑤
},
```

① Kick off the Ajax bar by setting it to 10 percent

② Set the `onUploadProgress` callback to increment the Ajax bar whenever there are updates. We multiply the `loaded` value (between 0.0 and 1.0) by 80

③ Bump up to 20 percent complete

④ Signal the Ajax bar that the operation is complete

⑤ If the Ajax bar is not already 100 percent, put it there now

How Does This Work?

The `ref` to progressBar gives us access to the `start`, `stop`, and `increment` methods of `QAjaxBar` and we are passing the `increment` and `stop` methods to our `loadTodos` method in the store. The store action can then manipulate the state of the `QAjaxBar` based on callbacks from the Axios REST client.

Error Handling

Inside of our store, we need to handle potential errors when making calls to the REST API. Quasar has us covered with it's **Notify** plugin

- Open `src/quasar-user-options.ts` and add the `Notify` plugin

```
import './styles/quasar.sass'
import '@quasar/extras/material-icons-round/material-icons-round.css'
import '@quasar/extras/mdi-v4/mdi-v4.css'
import '@quasar/extras/material-icons/material-icons.css'

// To be used on app.use(Quasar, { ... })
export default {
  config: {},
  plugins: {
    'Notify'
  }
}
```

- Open `src/stores/TodoStore.ts` and we can add notifications to our action method:

```
async loadTodos(
                notify: (message: string, type: string) => void,
                increment: (increment?: number) => void,
                stop: () => void
                ) {
    increment(10);
    const axiosConfig = {
        onUploadProgress: (progressEvent: ProgressEvent) => {
            increment(progressEvent.loaded * 80);
        }
    }
    increment(20);
    try {
        const { data } = await this.todoApi.getAllTodos(axiosConfig);
        this.updateTodos(data);
        stop();
    } catch (err) {
        notify('An error occurred loading Todo items from the API', 'negative');
    }
    increment(100);
},
```

- Then we need to add the changes to `Home.vue`

```
import { todoStore } from "@/stores/TodoStore"
import { QAjaxBar, useQuasar } from 'quasar';          ①
import { computed, defineComponent, ref } from "vue";

export default defineComponent(() => {
  const todos = todoStore();
  const progressBar = ref<QAjaxBar>();
  const increment = (increment?: number) => progressBar?.value?.increment(increment);
  const stop = () => progressBar?.value?.stop();
  const $q = useQuasar();                              ②
  const notify = (message: string, type = 'info') => $q.notify({ message, type });  ③

  function loadTodos() {
    progressBar?.value?.start();
    todos.loadTodos(notify, increment, stop);          ④
  }
```

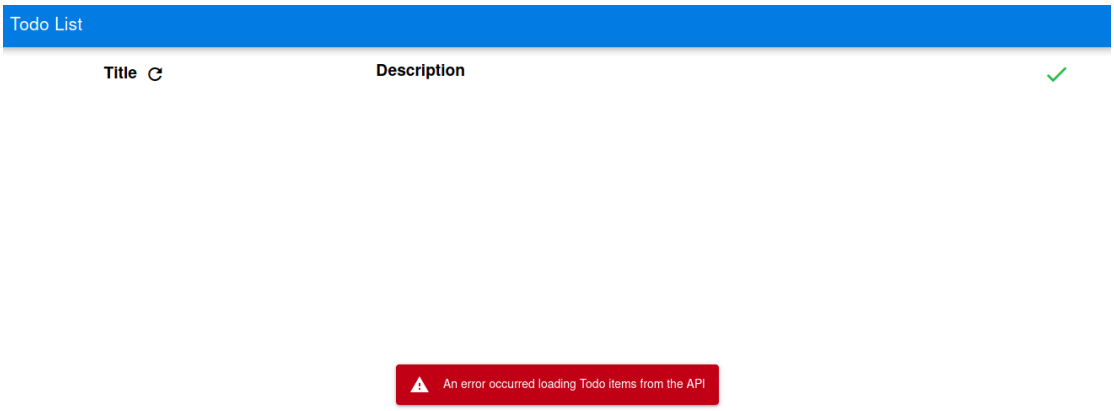
① Import the `useQuasar` function

② Instantiate the `$q` Quasar helper object

③ Create an anonymous function for notifications we can pass to our store

④ Update the call to `loadTodos`, passing in the notify function

If we stop our `yarn watch` and instead just launch the Vue application with `yarn serve`, we can reload the page and see an error when the API call fails



And Now The Fun Begins

So far, I have shown you some nice features of being able to have a Mock API and using Quasar Framework to build a UI. Now, we combine those 2 capbilities in order to really accellerate our ability to deliver business value. Let us imagine that we show our simple Todo application to our business stakeholder and they respond "But that's missing the features I need like a due date and a completion indicator!". In a traditional application development environment, you would have to run to tell the backend developers to make changes while the UI is updated as well. Since we have not yet involved any backend developers, we do not really care. We just make a quick change to our API contract and use those new fields in our UI! We continue iterating with feedback from our stakeholders until we achieve the user experience they desire. Only **AFTER** we have the user experience defined and validated do we then use the API contract to generate most of the backend and therefore we save time and rework. In other words, we deliver busines value more efficiently.

- Start by adding the new fields to the API contract

```
components:
  schemas:
    NewTodo:
      type: object
      required:
      - title
      properties:
        title:
          type: string
          maxLength: 255
        description:
          type: string
        id:
          type: string
          format: uuid
        due_date:              ①
          type: string
          format: date-time
          nullable: true
        completed:             ②
          type: string
          format: date-time
          nullable: true
```

① The new `due_date` field which is nullable

② The new `completed` field which is nullable

- Add some markup to our template inside of `Home.vue`

```
<div :class="headerClasses">
  <div class="small-cell" />
  <div class="col-grow">
    Title/Description
    <q-btn icon="refresh" dense flat @click="reload" />
  </div>
  <div class="col-2">Due</div>
  <div class="small-cell" style="text-align: left;">
    <q-icon name="check" color="positive" size="md"/>
  </div>
</div>
<q-scroll-area style="height: 82vh; width: 100vw;">
  <q-list>
    <q-item v-for="todo in todoList" :key="todo.id" class="row datatable">
      <q-item-section class="small-cell" dense>
        <q-btn icon="edit" size="0.8rem" flat dense />
      </q-item-section>
      <q-item-section class="col-grow">
        <q-expansion-item :label="todo.title">
        <template v-slot:header>
          <span class="title">{{ todo.title }}</span>
        </template>
        <template v-slot:default>
          <span class="description">{{ todo.description }}</span>
        </template>
        </q-expansion-item>
      </q-item-section>
      <q-item-section class="col-2">
        {{ dateFormat(todo.due_date) }}
      </q-item-section>
      <q-item-section class="small-cell">
        <q-checkbox :model-value="isComplete(todo.completed)" dense flat />
      </q-item-section>
    </q-item>
  </q-list>
</q-scroll-area>
```

SUCCESS - You are immediately able to see the new information in your development webapp! While live reloading is not that uncommon, remember that we are accessing and loading that data **from a REST API server** already. Once we iteratively refine this UI and user experience, we can hand off the OpenAPI contract to the backend developers for a very efficient implementation which will have **little or no integration issues!**