Get Started with Ionic

Prerequisites:

1. Node.js and npm:

Make sure you have Node.js and npm (Node Package Manager) installed on your machine. You can download and install them from nodejs.org.

1. Ionic CLI:

Install the Ionic CLI globally using the following command:

```
$ npm install -g @ionic/cli
```

Step 1: Create a new Ionic project.

Although the framework offers an App Creation Wizard, this time we will create the whole project using the terminal.

With that said, open your terminal and run the following command to create a new Ionic project:

```
$ ionic start
```

As the command starts to run, you'll be asked the following questions:

```
? Use the app creation wizard? No

? Framework? Vue

? Project name? tutorial_project

? Starter template: tabs
```

It may need a few minutes until the command finishes running and the project is created.

If you want to skip the whole questioning process that comes with the command stated before, you can also compress it to be faster.

```
$ ionic start tutorial_project tabs --type=vue
```

Step 2: Navigate to the project directory.

```
$ cd tutorial_project
```

Step 3: Serve your app.

Run the following command to start a local development server and see your app in the browser:

```
$ ionic serve
```

Open your browser and navigate to http://localhost:8100. You should be able to see your app with the simple tabs template.

Step 4: Explore the project structure.

Open the project in your preferred code editor. The important files and directories are mostly located in the *src* folder.

To start the project, make sure the current files are like this: "src/views/Tab1Page.vue"

```
<template>
  <ion-page>
    <ion-header>
      <ion-toolbar>
        <ion-title>Hello World</ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content :fullscreen="true">
      <ion-header collapse="condense">
        <ion-toolbar>
          <ion-title size="large"> Hello World <ion-title>
        </ion-toolbar>
      </ion-header>

      <ExploreContainer name="Hello World Page" />
    </ion-content>
  </ion-page>
</template>

<script setup lang="ts">
import { IonPage, IonHeader, IonToolbar, IonTitle, IonContent } from
'@ionic/vue';
</script>
```

"src/views/Tab2Page.vue"

```
<template>
```

```
  <ion-page>
    <ion-header>
      <ion-toolbar>
        <ion-title>Todo-app</ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content :fullscreen="true">
      <ion-header collapse="condense">
        <ion-toolbar>
          <ion-title size="large">Todo-app</ion-title>
        </ion-toolbar>
      </ion-header>

      <ExploreContainer name="Todo-app page" />
    </ion-content>
  </ion-page>
</template>

<script setup lang="ts">
import { IonPage, IonHeader, IonToolbar, IonTitle, IonContent } from
'@ionic/vue';
</script>
```

“src/views/TabsPage.vue”

```
<template>
  <ion-page>
    <ion-tabs>
      <ion-router-outlet></ion-router-outlet>
      <ion-tab-bar slot="bottom">
        <ion-tab-button tab="tab1" href="/tabs/tab1">
          <ion-icon aria-hidden="true" :icon="square" />
          <ion-label>Hello World</ion-label>
        </ion-tab-button>

        <ion-tab-button tab="tab2" href="/tabs/tab2">
          <ion-icon aria-hidden="true" :icon="ellipse" />
          <ion-label>Todo-app</ion-label>
        </ion-tab-button>
      </ion-tab-bar>
    </ion-tabs>
  </ion-page>
</template>

<script setup lang="ts">
```

```
import  {  IonTabBar,  IonTabButton,  IonTabs,  IonLabel,  IonIcon,  IonPage,
IonRouterOutlet } from '@ionic/vue';
import { ellipse, square } from 'ionicons/icons';
</script>
```

Now, if you want, you're free to delete the file "src/views/Tab3Page.vue" and "src/components/ExploreContainer.vue", as we won't be using them in this tutorial. Remember that you also need to delete the code that calls the Tab3Page on the file "src/router/index.ts".

```
      {
        path: 'tab3',
        component: () => import('@/views/Tab3Page.vue')
      }
```

Step 5: Hello world Page

Open the file "src/views/Tab1Page.vue" and insert the following code within the <ion-content> component:

```
      <ion-fab vertical="bottom" horizontal="center" slot="fixed">
        <ion-fab-button id="hello-alert">
          <ion-icon :icon="handLeft"></ion-icon>
        </ion-fab-button>
      </ion-fab>

      <ion-alert
        trigger="hello-alert"
        header="Hello World!"
        sub-header="Welcome to the Ionic Vue app."
        :buttons="alertButtons"
      ></ion-alert>
```

To incorporate a button within the ion-alert, include the following line in the <script> component:

```
import { handLeft } from 'ionicons/icons';

const alertButtons = ['Thank you!'];
```

Now execute the app and test it by clicking the "Hello" button to ensure the alert is displayed.

This code is designed to display a popup on the center of the screen. Clicking this button activates the ion-alert, presenting a message with the header “Hello World!” and a sub-header “Welcome to the Ionic Vue app.”. The alert includes a “Tank you!” button, to dismiss the popup.

Step 6: Todo-app

Create two components on “src/components” folder: Task.vue, and NewTask.vue.
On this app, we will use the terms components that have various purposes in the development of web applications, contributing to the overall structure, modularity, and maintainability of the codebase. Here's a breakdown of what these components are for. In case for more deep understanding, make sure to view this link:

- https://vuejs.org/guide/essentials/component-basics.html

Start to write code in the Parent (Tab2Page.vue) that will communicate with the child components (Task.vue, and NewTask.vue).
Write the template HTML like this:

```
<template>
  <ion-page>
    <ion-header>
      <ion-toolbar>
        <ion-title>Todo-app</ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content :fullscreen="true">
      <ion-header collapse="condense">
        <ion-toolbar>
          <ion-title size="large">Todo-app</ion-title>
        </ion-toolbar>
      </ion-header>

      <div class="container">
        <div>
          <ion-item class="ion-padding">
            <ion-row>
              <ion-col>
                <h4>Pending: {{ pending }}</h4>
                <h4>Completed: {{ completed }}</h4>
              </ion-col>
              <NewTask @AddTask="addTask"></NewTask>
            </ion-row>
          </ion-item>
          <hr />
          <br />
          <ul class="list-group">
            <Task
              v-for="task in tasks"
              :key="task"
              :task="task"
              :readonly="false"
              @RemoveTask="removeTask"
              @ToogleComplete="toggleCompleted"
            ></Task>
          </ul>
          <hr />
          <h4>Completed Tasks</h4>
          <ul class="list-group">
            <task
              v-for="task in completedTasks"
              :key="task"
              :task="task"
              :readonly="true"
            >
            </task>
          </ul>
          <hr />
          <h4>Incompleted Tasks</h4>
          <ul class="list-group">
            <task
              v-for="task in incompletedTasks"
              :key="task"
              :task="task"
              :readonly="true"
            ></task>
          </ul>
        </div>
      </div>
    </ion-content>
  </ion-page>
</template>
```

The template section of the code defines the visual structure of the "Tab2Page" component using Ionic components. Here's a breakdown:

- **<ion-page>**: Represents a page in an Ionic application.
- **<ion-header>**: Contains the header for the page.
  - **<ion-toolbar>**: Displays the title of the app.
- **<ion-content>**: Contains the main content of the page.
  - **<div class="container">**: Wraps the content for styling purposes.

- **<ion-item>**: Displays the count of pending and completed tasks.
- **<NewTask @AddTask="addTask"></NewTask>**: Uses the custom "NewTask" component to add new tasks.
- **<ul class="list-group">**: Displays a list of tasks using the custom "Task" component.
- **<hr />**: Separates different sections of the task list.
- **<h4>Completed Tasks</h4>**: Displays a header for the completed tasks section.
- **<h4>Incompleted Tasks</h4>**: Displays a header for the incompleted tasks section.

Now, continue to write the script that is responsible to control the communication of the child components and functions:

```
<script setup lang="ts">
import {
  IonPage,
  IonHeader,
  IonToolbar,
  IonTitle,
  IonContent,
} from "@ionic/vue";

import { ref, computed } from "vue";
import NewTask from "./../components/NewTask.vue";
import Task from "./../components/Task.vue";

const tasks = ref([]);

const addTask = (newTask: any) => {
  if (newTask) {
    tasks.value.push(newTask);
  }

  console.log(newTask);
};

const removeTask = (task: any) => {
  const idx = tasks.value.findIndex((element) => element === task);
  if (idx >= 0) {
    tasks.value.splice(idx, 1);
  }
};

const toggleCompleted = (task: any) => {
  task.completed = !task.completed;
};

const completed = computed(() => {
  return tasks.value.reduce(
    (acumulador, task) => (task.completed ? acumulador + 1 : acumulador),
    0
  );
});

const pending = computed(() => {
  return tasks.value.reduce(
    (acumulador, task) => (!task.completed ? acumulador + 1 : acumulador),
    0
  );
});

const completedTasks = computed(() => {
  return tasks.value.filter((t) => t.completed);
});

const incompletedTasks = computed(() => {
  return tasks.value.filter((t) => !t.completed);
});
</script>
```

The script section contains the logic for handling tasks. Here's what each part does:

- **<script setup lang="ts">**: Specifies that this is a composition API script setup using TypeScript.
- **import {...}**: Imports necessary components and functions from Ionic and Vue.
- **const tasks = ref([]);**: Creates a reactive variable to store the list of tasks.
- **const addTask = (newTask: any) => {...};**: Adds a new task to the list.
- **const removeTask = (task: any) => {...};**: Removes a task from the list.
- **const toggleCompleted = (task: any) => {...};**: Toggles the completion status of a task.
- **const completed = computed(() => {...});**: Computes the count of completed tasks.
- **const pending = computed(() => {...});**: Computes the count of pending tasks.
- **const completedTasks = computed(() => {...});**: Computes the list of completed tasks.
- **const incompletedTasks = computed(() => {...});**: Computes the list of incompleted tasks.

Now, move on to the child component “NewTask.vue”. This child will be responsible to add new tasks to be showing below in the list.
Write the template HTML like this:

```
<template>
  <ion-form class="ion-padding">
    <ion-fab horizontal="end" slot="fixed">
      <ion-fab-button id="add-todo">
        <ion-icon :icon="add"></ion-icon>
      </ion-fab-button>
    </ion-fab>

    <ion-alert
      trigger="add-todo"
      header="Please enter your info"
      :buttons="alertButtons"
      :inputs="alertInputs"
      @didDismiss="onDismiss"
    >
    </ion-alert>
  </ion-form>
</template>
```

The template section of the code defines the visual structure of the "NewTask" component using Ionic components. Here's a breakdown:

- **<ion-form>**: Represents a form in an Ionic application.
    - **<ion-fab>**: Represents a floating action button container.
        - **<ion-fab-button>**: Displays a button for triggering the alert.
            - **<ion-icon :icon="add"></ion-icon>:** An add icon for the "+" sign.
- **<ion-alert>**: Represents an alert dialog.
    - **trigger="add-todo"**: Associates the alert with the specified trigger button.
    - **:buttons="alertButtons"**: Specifies buttons for the alert.
    - **:inputs="alertInputs"**: Specifies input fields for the alert.
    - **@didDismiss="onDismiss"**: Listens to the dismissal event of the alert.

Script:

```
<script setup lang="ts">
import { ref, defineEmits } from "vue";
import { add } from "ionicons/icons";

const newTask = ref("");

const emit = defineEmits(["AddTask"]);

const alertButtons = [
  {
    text: "Cancel",
    role: "cancel",
    handler: () => {
      console.log("Alert Cancel");
    },
  },
  {
    text: "Ok",
    role: "ok",
    handler: () => {
      console.log("Alert Ok");
    },
  },
];

const alertInputs = [
  {
    type: "textarea",
    placeholder: "Give a description of the task",
  },
];

const onDismiss = (event: CustomEvent) => {
  if (event.detail.role === "ok") {
    newTask.value = event.detail.data.values[0];
    addTask();
  }
};

const addTask = () => {
  if (newTask.value.trim()) {
    emit("AddTask", {
      description: newTask.value,
      completed: false,
    });
  }
  newTask.value = "";
};

const onTaskChange = (event) => {
  newTask.value = event.detail.value;
};
</script>
```
The script section contains the logic for handling new tasks. Here's what each part does:

- **<script setup lang="ts">**: Specifies that this is a composition API script setup using TypeScript.
- **import {...}**: Imports necessary components and functions from Vue.
- **const newTask = ref('');**: Creates a reactive variable to store the description of the new task.
- **const emit = defineEmits(['AddTask']);**: Defines the emit for adding a new task.
- **const alertButtons = [...]**: Configures buttons for the alert, including handling the "Ok" and "Cancel" actions.
- **const alertInputs = [...]**: Configures input fields for the alert, including a textarea for the task description.
- **const onDismiss = (event: CustomEvent) => {...};**: Handles the dismissal of the alert, extracts the entered task description, and triggers the "AddTask" event.
- **const addTask = () => {...};**: Emits the "AddTask" event with the new task details.
- **const onTaskChange = (event) => {...};**: Handles changes in the task input field.

In Tab2Page.vue, notice how the NewTask component emits an event (AddTask) to notify the parent about a new task. The parent, in turn, handles this event by adding the new task to the task list.

The Task.vue component represents an individual task. It allows users to mark tasks as completed, remove tasks, and edit task descriptions. The component emits events (RemoveTask and ToggleComplete) to communicate with the parent Tab2Page.

Template:

```
<template>
  <ion-item class="ion-padding">
    <span :class="{ completed: task.completed }">{{ task.description
}}</span>

    <ion-buttons slot="end" v-show="!readonly">
      <ion-button @click="toggleCompleted(task)" v-if="task.completed">
        <ion-icon :icon="repeat"></ion-icon>
      </ion-button>
      <ion-button @click="toggleCompleted(task)" v-else>
        <ion-icon :icon="checkmark"></ion-icon>
      </ion-button>

      <ion-button color="danger" @click="removeTask(task)">
        <ion-icon :icon="trash"></ion-icon>
      </ion-button>

      <ion-button @click="editTask(task)">
        <ion-icon :icon="create"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-item>
  <div v-if="taskEdit">
    <ion-alert
      :is-open="isAlertOpen"
      header="Edit Task"
      :inputs="alertInputs"
      :buttons="alertButtons"
      @didDismiss="closeEdit"
    ></ion-alert>
  </div>
</template>
```

The template section of the code defines the visual structure of the task item using Ionic components. Let's break it down:

- **<ion-item>**: Represents an item in an Ionic list.
- **<span>**: Displays the task description with a class conditionally applied based on completion status.
- **<ion-buttons>**: Contains buttons for actions like toggling completion, removing, and editing the task.
- **<ion-button>**: Used for various actions like completing, removing, and editing the task.
- **<ion-alert>**: A modal component for editing the task, displayed conditionally when the **taskEdit** variable is true.

Script:

```
<script setup lang="ts">
import { ref, defineProps, defineEmits } from "vue";
import { repeat, checkmark, trash, create } from "ionicons/icons";

const taskEdit = ref(null);
const props = defineProps(["task", "readonly",]);

const task = props.task;



const emit = defineEmits(["RemoveTask", "ToogleComplete"]);

const removeTask = (task) => {
  emit("RemoveTask", task);
};

const toggleCompleted = (task) => {
  emit("ToogleComplete", task);
};

const alertInputs = [
  {
    name: "description",
    type: "text",
    placeholder: "Enter the task description",
    value: task.description,
  },
];

const alertButtons = [
  {
    text: "Cancel",
    role: "cancel",
    handler: () => {
      console.log("Cancel clicked");
    },
  },
  {
    text: "Save",
    handler: (data) => {
      task.description = data.description;
    },
  },
];

const isAlertOpen = ref(false);

const editTask = (task) => {
  taskEdit.value = task;
  isAlertOpen.value = true;
};

const closeEdit = () => {
  taskEdit.value = null;
};
</script>
```

The script section contains the logic for handling tasks. Here's what each part does:

- **<script setup lang="ts">**: Specifies that this is a composition API script setup using TypeScript.
- **import { ref, defineProps, defineEmits } from "vue";**: Imports necessary functions from Vue for creating reactive variables and defining props and emits.

- **const taskEdit = ref(null);**: Creates a reactive variable to store the task being edited.
- **const props = defineProps(["task", "readonly"]);**: Defines props to receive task and readonly status.
- **const task = props.task;**: Assigns the task prop to a variable for easier access.
- **const emit = defineEmits(["RemoveTask", "ToogleComplete"]);**: Defines emits for removing and toggling completion of tasks.
- **const removeTask = (task) => { emit("RemoveTask", task); };**: Emits the "RemoveTask" event when the removeTask function is called.
- **const toggleCompleted = (task) => { emit("ToogleComplete", task); };**: Emits the "ToogleComplete" event when the toggleCompleted function is called.
- **const alertInputs = [...]**: Configures the input fields for the edit task modal.
- **const alertButtons = [...]**: Configures buttons for the edit task modal, including handling the save action.
- **const isAlertOpen = ref(false);**: Creates a reactive variable to control the visibility of the edit task modal.
- **const editTask = (task) => { taskEdit.value = task; isAlertOpen.value = true; };**: Sets the task being edited and opens the edit task modal.
- **const closeEdit = () => { taskEdit.value = null; };**: Clears the task being edited and closes the edit task modal.

Step 7: Mobile Deployment.

If you’re ready to deploy your ionic application on your mobile device, then these are the steps that you need to take.

We’ll start by rebuilding the project, so if you’re currently running it then you’ll need to stop.
To build the application, we have the command:

```
$ ionic build
```

After that is done, you’ll need to run the following command:

```
$ ionic cap add android
$ ionic cap add ios
```

This will create both Android and iOS folders at the root of the project, which are entirely standalone native projects that should be considered part of you Ionic app.

Important: Every time you perform a build (e.g. ionic build) that updates your web directory (default: build), you'll need to copy those changes into your native projects.

To open the project on Android Studio (platform on which we will test the native capabilities of ionic in this tutorial), run the following command:

```
      $ ionic cap open android
```

Note: Android Studio need to be installed in order for this to work, otherwise testing your app on your android device won't be possible with this setup (same thing for iOS but with XCode).

It will take a couple minutes for the gradle to load. After that run the following command to test the app:

```
      $ ionic cap run android -l --external
```

When you run this command you will be asked the following questions:

```
      ? Which device would you like to target?
      ? Please select which IP to use:
```

The Live Reload server will start up, and the native IDE of choice will open if not opened already. Within the IDE, click the Play button to launch the app onto your device. However, do not run the app on Android Studio until the commands finishes running, which means you should wait until you see this message on the terminal:

```
      [INFO] Development server running!
```