Advanced Web Programming

Lab7 - Start to Create SPA App in Vue 3

In previous Vue labs, you learned how to use components, how to communicate between components, and you learned about the binding methods between component logic and HTML code. You know what directives are and how to use them correctly.

In lab 7 we will want to combine all these elements to create typical CRUD applications. It will be SPA application. As we do not use the backend yet, the data for our application will be defined on the frontend side as a local model.

We will build a Vue.js front-end Student Comments Application in that:

- Each comment has id, author (author's name), title and description (comment contents).
- We can show or hide a contents each comment.

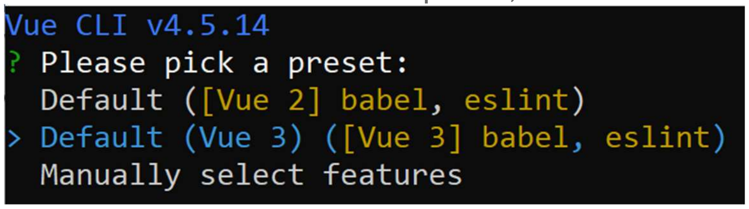
Lets start the project.

Step 1. Preparing the Project

Open cmd at the folder you want to save Project folder, run command:

```
vue create efrai-student-comments
```

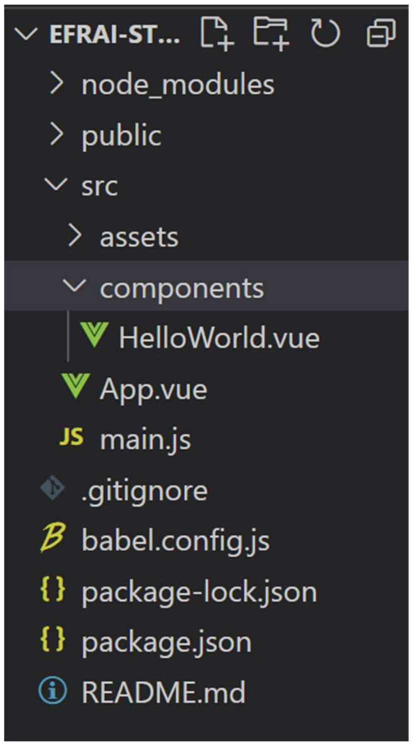
You will see some options, choose **Default ([Vue 3] babel, eslint)**.



```
Vue CLI v4.5.14
? Please pick a preset:
  Default ([Vue 2] babel, eslint)
> Default (Vue 3) ([Vue 3] babel, eslint)
  Manually select features
```

After the process is done.

After we open the project with a code editor, we are going to notice the following folder structure:



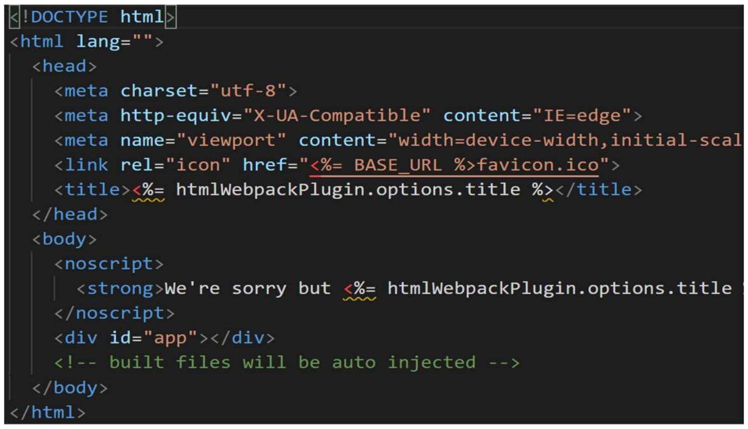
I remind you (because we talked about it in lab 6) the most important files and folders:
src – Our project source code:

- **assets** – Module assets which will be processed with Webpack
- **components** – This is where we keep our UI components
- **App.vue** – This is an entry point component, it's the main UI component in which all the other components will render
- **main.js** – Entry point file which will mount App.vue – our main UI component

public – pure assets that will not be processed with Webpack

- index.html – You may remember that the SPA application always rewrites the content of one file, so, this is that file. This is the file that we are serving to our visitors. After building a project, this file will load static files that were bundled with Webpack.

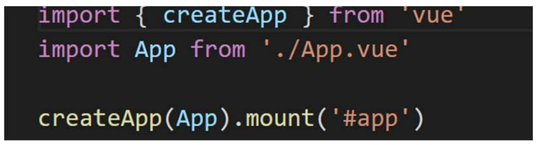
Let's take a look at the index.html file:



```html
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scal
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title 
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

The only important thing in this file is a div tag with id app, which is going to be replaced with our App.vue component. On the next line, we can notice the comment that says: “built files will be auto injected”. So, our js file, which is bundled with Webpack, is going to be injected below the div with id app.

You may remember that main.js is our entry point, right? Let’s take a look at main.js file which is the first file to be executed when the application starts:

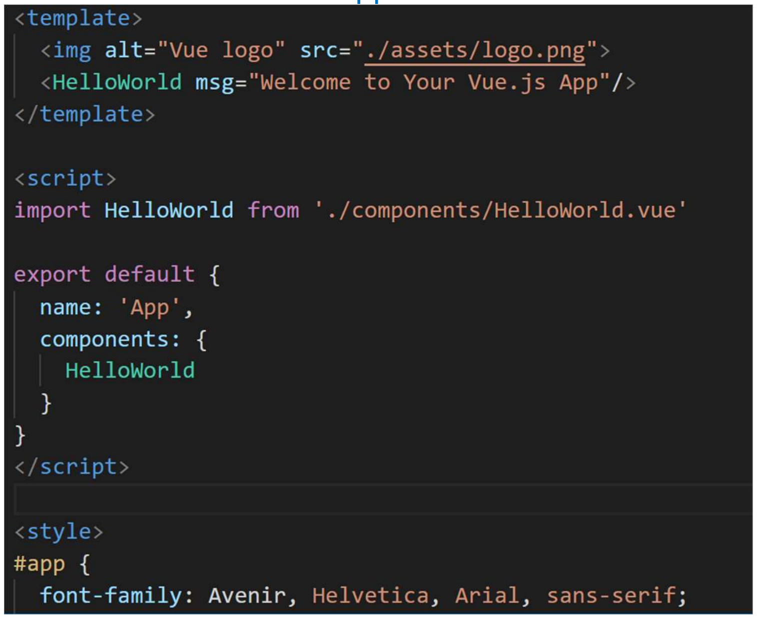


```
import { createApp } from 'vue'
import App from './App.vue'

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

In this code example, we can notice that our main.js imports the createApp module. In the next line, we can see additional import. It is our App component (App.vue).
We instantiate a new Vue component by using CreateApp which is going to mount our App.vue component inside a div with the id app attribute – you remember div with id app in index.html?

Let’s take a look at the App.vue file:



```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld msg="Welcome to Your Vue.js App"/>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
```

This component acts as a container - root components for other (future) components in the application.

Every Vue component may contain a template, a script, and a style section.
The template is a visible content of the component. The script is a logic of the component Style, you as you know, is a style for a template

Template
In a template, we can use native HTML tags as well as our custom Vue components.

Script
In the script tag, we write logic for that component. In this component, we only have the name property because this component isn't dynamic, it is nothing other than the wrapper.

Style
Here we can write a style for a template. Style can be scoped or global. The scoped style is written this way: <style scoped>. That type of style only affects the template of that component and root element of child components so there is no way for style to leak to another component. Counterwise, a non-scoped style will be shared between components.

Step 2. Add Vue Router to Vue 3

In Vue.js, we are already composing our application with components. We don't want all components to be displayed in one view. As you know, modern single-page apps, such as a Vue application can transition from page to page on the client-side (without requesting the server). We want to be able to switch between different views. That's why we need routing.
When adding Vue Router to the Vue app, all we need to do is map our components to the routes and let Vue Router know where to render them.
Vue Router is the official library for page navigation in Vue applications.

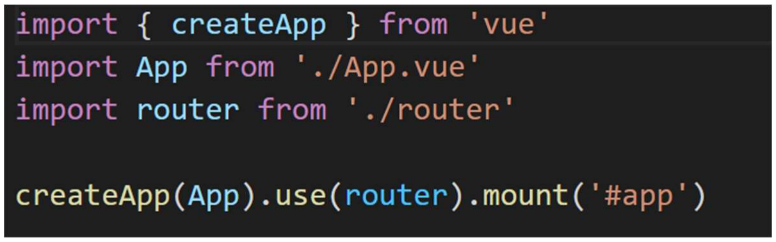
To add it to our project, run the following command in the project directory.

```
vue add router
```

The cli will ask if we want to use history mode. We'll be using history mode, so we'll select yes.

As you can see, the structure of our project has changed. There is new directory router in our app and some of files have changed.

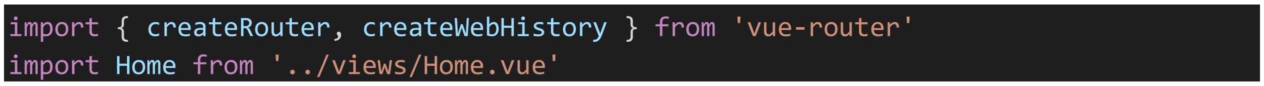
Open **src**/*main.js* and you see that app import the router in our project:



```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'

createApp(App).use(router).mount('#app')
```

Let's take a look at the router/index.js file:



```
import { createRouter, createWebHistory } from 'vue-router'
import Home from '../views/Home.vue'
```

```
const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',
    // route level code-splitting
    // this generates a separate chunk (about.[hash].js) for this route
    // which is lazy-loaded when the route is visited.
    component: () => import(/* webpackChunkName: "about" */ '../views/About.vue')
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
```

We create the routes as an array, each route has:

- path: the URL path where this route can be found.
- name: optional name to use when we link to this route.
- component: component to load when this route is called.

We also use createWebHistory to switch from using hash to history mode inside the browser, using the HTML5 history API.

As you see when we create router: a view directory is created containing the new components. This components represent view component in our app.

Let's open **src**/*App.vue*, this App component is the root container for our application, it will contain a navbar. Let's take a look at the App.vue file:

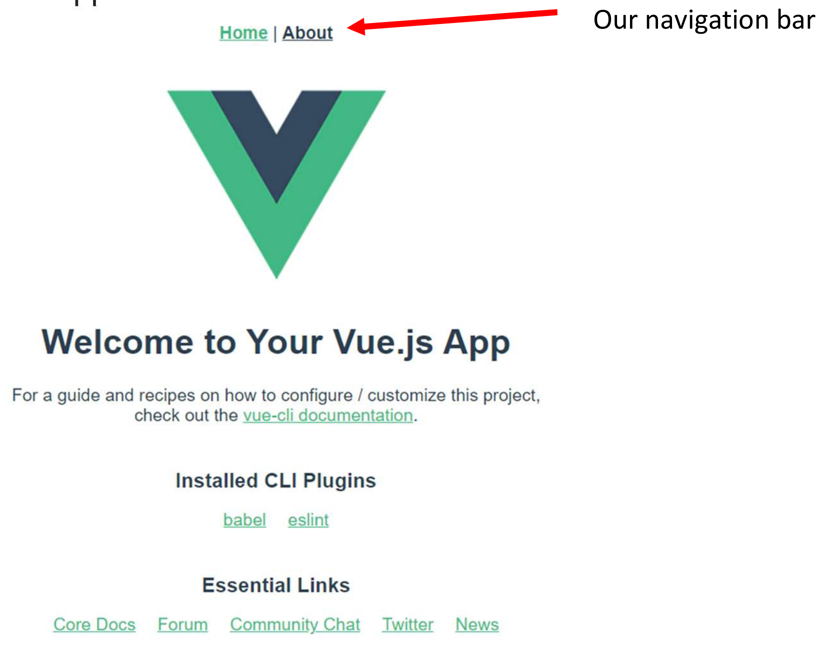
```
<template>
  <div id="nav">
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link>
  </div>
  <router-view/>
```

```
</template>
```

As you see template aour root component has changed too and has a navbar with two position. Here we can see also the <router-view/> element and that’s the place where our router is going to render a matching component for that route.

Now that we understand the code, it’s time to run the application and see everything on a live example.
Run: npm run serve

And our app looks like below:



Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.

Installed CLI Plugins

babel eslint

Essential Links

Core Docs Forum Community Chat Twitter News

We may notice the Vue.js logo which is the part of the App.vue component. Under the logo, we see the content of the HelloWorld.vue file which is rendered inside the <router-view/>element of the App.vue component. The application is rendering the HelloWorld.vue component because we are on the / route.

Step 3. Make Vuejs Components

In that step we create all component files (but without details implementation) and we will configure the navigation bar and organize the project structure.

In our project we should have 3 components:
CommentsList - component to manages list all comments
Comment, -   display only one comment

AddComment. – add new comment to list of comments

Let’s navigate to the src/components directory and create a new files and name their. For now, we are going to put some dummy text in the each component template. Next, we are going to make a route for that component and check if everything is OK.
So create that components:

```
// CommentsList.vue

<template>

   <div>

          CommentsList Component

    </div>

</template>



export default {

     name: 'CommentList'

 }
```

```
// Comment.vue

<template>

   <div>

          Comment Component

    </div>

</template>
```

```
export default {

     name: 'Comment'

 }
```

```
// AddComment.vue

    <template>

   <div>

         AddComment Component

    </div>

</template>

export default {

   name: 'AddComment'

 }
```

Creating a new Route

Let’s open the routes/index.js file and modify it:

```
import { createRouter, createWebHistory } from 'vue-router';
import AddComment from '../components/AddComment.vue';
import Comment from '../components/Comment.vue';
import CommentList from '../components/CommentList.vue';

const routes = [
  {
    path: '/',
    name: 'CommentList',
    component: CommentList
  },
  {
    path: '/AddComment',
    name: 'Add',
    component: AddComment
  }
]
```

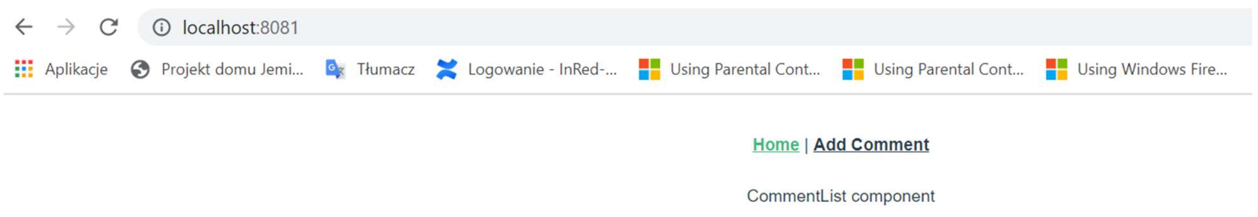
Look Good. We define new politics for routing.

Now we update App.vue file. In our navbar we want to have a direct link to CommentList and AddComment.

```
<template>
  <div id="nav">
    <router-link to="/">Home</router-link> |
    <router-link to="/AddComment">Add Comment</router-link>
  </div>
  <router-view/>
</template>

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

For example If we write in browser address correct value we have a

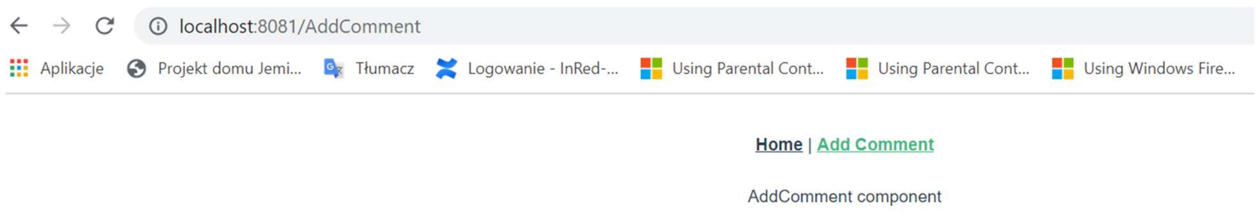

localhost:8081

Aplikacje | Projekt domu Jemi... | Tłumacz | Logowanie - InRed-... | Using Parental Cont... | Using Parental Cont... | Using Windows Fire...

Home | Add Comment

CommentList component

Or



AddComment component

But what if someone goes to URL which doesn’t exist,

like http://localhost:8080/#/whatever?

We need to display NotFound component when this happens. Let’s implement that.

Creating the NotFound Component

We are going to create the catch all route which will be triggered when none of our routes are matched.

Let’s first create the NotFound.vue component. In the src directory, we are going to make a new directory named error-pages. We are going to put

our NotFound.vue component which only shows the message like page is not found in that directory.

Here is how the ‘NotFound.vue’ file looks:

```
<template>
 <p>
  404 SORRY COULDN'T FIND IT!!!
 </p>
</template>
<script>
export default {
 name: 'NotFound'
};
</script>
<style scoped>
p {
  font-weight: bold;
  font-size: 50px;
  text-align: center;
  color: #f10b0b;
}
</style>
```

Now, we are going to open the routes/index.js and add the catch all route:

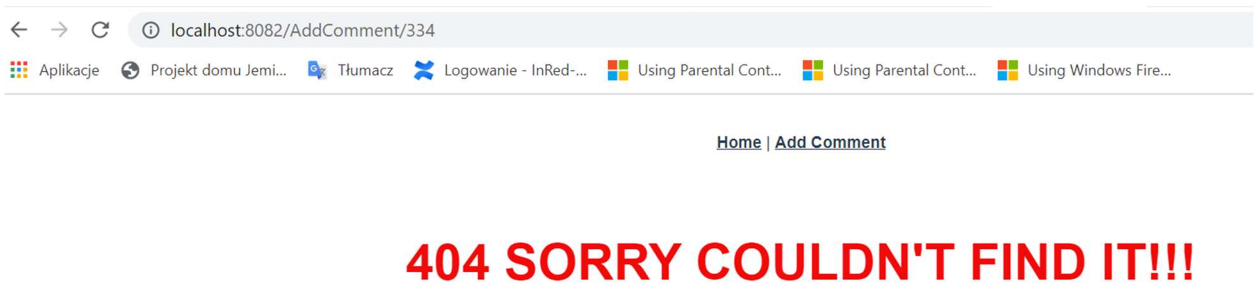
```
import { createRouter, createWebHistory } from 'vue-router';
import AddComment from '../components/AddComment.vue';
import Comment from '../components/Comment.vue';
import CommentList from '../components/CommentList.vue';
import NotFound from '../components/NotFound.vue';

const routes = [
  {
    path: '/',
    name: 'CommentList',
    component: CommentList
  },
  {
    path: '/AddComment',
    name: 'Add',
    component: AddComment
  },
  {
    path: '/:catchAll(.*)*',
    name: 'NotFound',
    component: NotFound
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
});
```

Let's inspect the result by navigating to for example:
http://localhost:port/AddComment/334


localhost:8082/AddComment/334

Home | Add Comment

404 SORRY COULDN'T FIND IT!!!

As you can see, creating our navigation menu and linking routes with components has been a pretty easy and straightforward job.

At the end this step we manage our project files. Now the file HelloWorld.vue is not used so we can to delete it from project. The same to the views files.

Step 4. Showing Fake Data

Now we want to implement CommentsList to display all comments. Because we don't have a backend for sending data we have to create data by own.

My Suggest is to create file for example: FakeDataService.js which have an array 6 object items.
I remind you that the object consists of: id (id number), author (author name), title, description (comments note) and hide (true or false for hide description field).

Exercise 1. Define FakeDataService.

Let our fakeDataService have the following methods:

```
 getAll() {


}
   find (author)  {
 },
 create(data) {
  },
 delete(id) {
 }
};
```

Implement FakeDataService.

Exercise 2. Fatching data from FakeDataService and Display in Component CommentsList .

To implement display single comment use Comment component. Let Comment component will be child of CommentsList. Using prop to inject data to Comment. Additionally, implement the functionality of the show / hide button - which hide/show description in comment.

Exercise 3. For all items in ComponentsList add button delete.

Implement this functionality: when you delete button, select item from ComponentList should be delete form List. Let buttons belongs to Comment.

Exercise 4. Implement addComment

In that component we have a form to add new data to fake Comment Data.

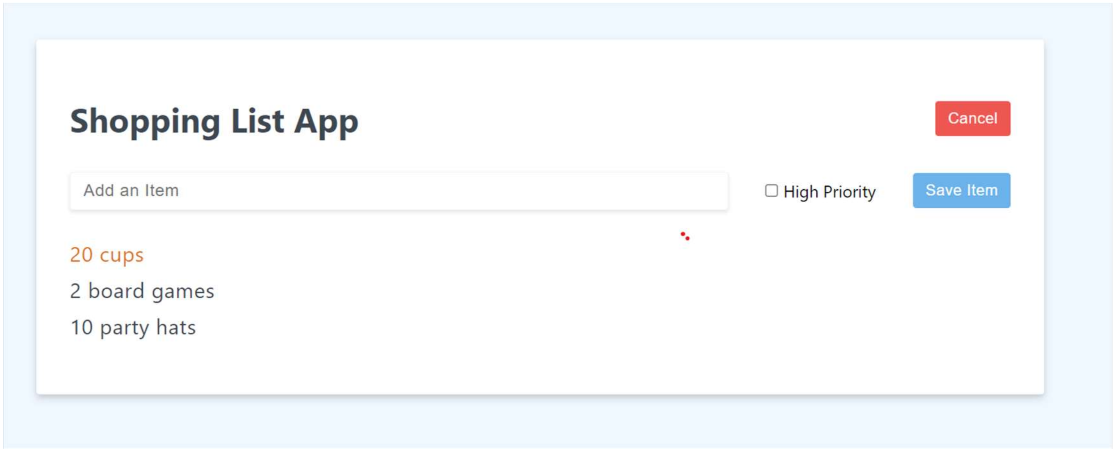
When you add data and click save new data will be added to FakeDataService.

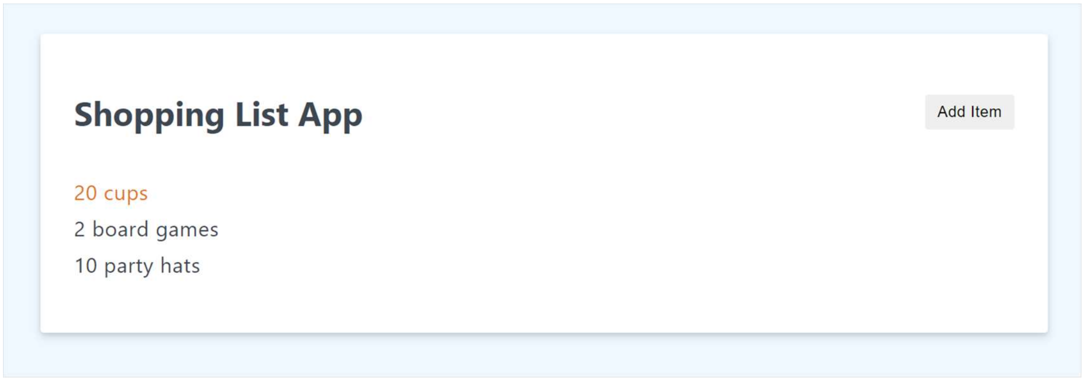
Exercise 5. In ComponentsList implement Search bar for finding comments by authors.

Homework.

Deadline time 7.04.2022 23:59

Exercises 1. Create shopping list app. Goal app is to manage shopping list. You can add new items to shopping list or disable this possibility (example of screen from example app you can seen below). Button save item add to list data from text input. Checkbox hight priority set level od priority added item. When checkbox is seleted items is added in red color (as a very important position in the list). When you click button cancel possibility of add items to list is off. Then you can see only Add item button. When you click this button your app display first screen and enable to add items to list.





Exercises 2. Create shop application. Let our application have three views: home, products, shopping list.

Let view: home display some information about your shop. It is a some text and photo a few slides presenting our store.

View products display collection of products from our store with the option of purchasing them. The third view represent details your basket. App display a list of the products you have purchased with their quantity. Below is a price summary

In application you have a collection at least 10 elements. Every element has name, number of items, unit price and url of product photo. Display list of products. In one row we have photo, name, price and number of product items. There should be 2 buttons + and - next to each product, allowing you to add a product to the basket list or return it. When you add item of product to basket number of product is decrement. If you return item from basket to list number of product increment.

Basket in product view display only the value of the products ordered. Detailed list of purchased items is display in shopping list view.

If the value of the product in the basket is 0, a different message should be displayed than when the number of available products is greater than 0.

If the amount of the product drops to zero, the - button should be hidden. After all, we do not want to reserve a product that we no longer have.

When the amount of the product is approaching 0 (e.g. from 3 downwards), it should be marked in a graphic way, e.g. a different background, font color, font size or other visual way.

Similarly, a distinction should be made between the product with the lowest unit price and the highest - with an additional border covering a given product - green - the most expensive, red - the cheapest.

Also display the total number of currently available products - if it is more than 10, should be displayed on a green background, if below 10 - on a red background.