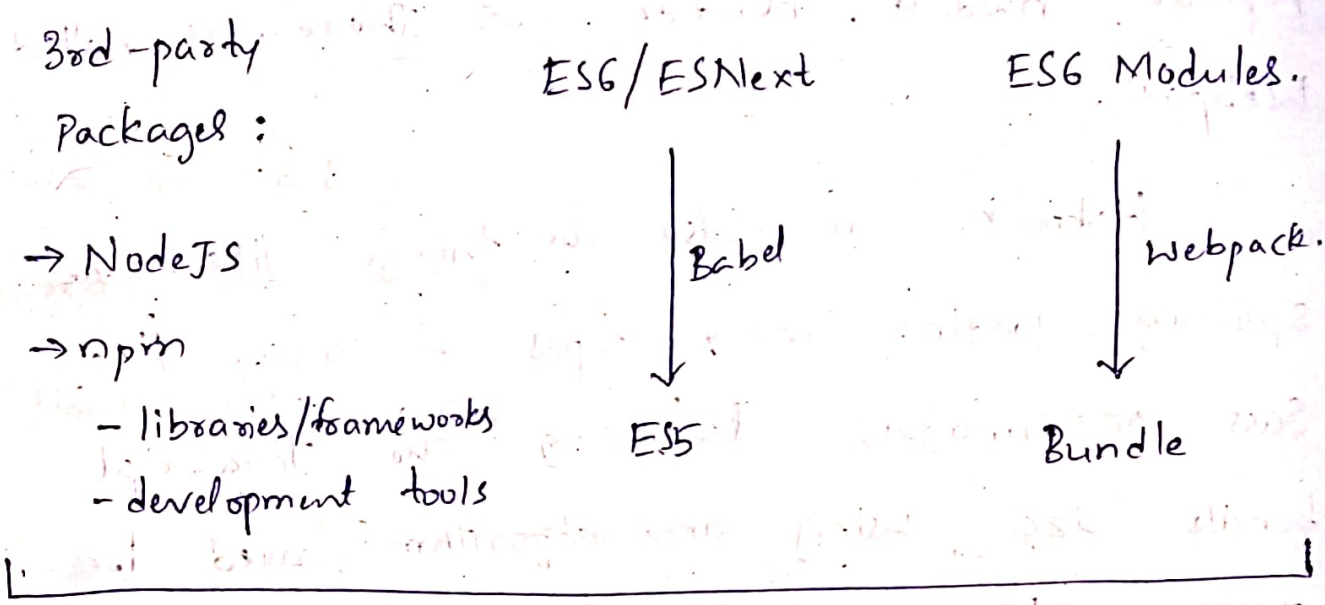
Chapter 9: Modern JavaScript Using ES6, NPM, Babel & Webpack:.

Modern JavaScript: A Brief Overview:



Putting it all together with an automated development setup powered by npm scripts.

Babel: Babel converts ES6, ES7 or ES8 to ES5. So all browsers can be able to understand our code.

ES6 Modules: In order to make our code more modular, we use ES6 modules. By separating different parts of our app into different files. But the problem with these modules

is that, right now, browsers don't support this functionality yet. So we have to bundle these modules together into a single file with module bundler. The most popular module bundler out there is called Webpack.

Webpack can also do things like code splitting, loading many types of assets like Sass or images, decreasing our JavaScript bundle size using an algorithm called tree shaking etc.

5: A Modern Setup: Installing Node.js & NPM:

First install Node.js.

To check if it is installed correctly, use this command:

```
node -v ↵
```

This shows the version if it is installed.

Then check for npm:

```
npm -v ↵
```

Now, we have to create package.json for our project. First navigate into our project folder and create package.json.

Type:

npm init ↵

It will ask some questions. Fill the required and click enter.

Now let's install Webpack using npm.

npm install webpack --save-dev ↵

The --save-dev will save webpack as a development dependency of our project. This will appear in our package.json.

We basically have two types of Node.js packages :→ Libraries → Frameworks.

→ Development Tools.

So libraries like React or jQuery are real dependencies because we use the code in our project. But something like webpack is just

a development tool. So it will be in the devDependencies in package.json. Whereas libraries or frameworks like jQuery or React will be in the dependencies and we use --save for them.

If we share our entire project with someone else, we wouldn't have to share all the huge folders like node_modules. We can simply share the package.json file which has the information about our dependencies and installs them automatically. The other user just need to type this:

```
npm install ↵
```

and all the dependencies used in package.json will be installed automatically.

To uninstall a package:

```
npm uninstall jquery --save
```

To install something globally:

```
npm install live-server --global.
```

6: A Modern Setup: Configuring Webpack:

Webpack is the most commonly used asset bundler.

Let's create a file called webpack.config.js in our project folder. In this fill, we will have one object in which we can specify our settings or configuration. Then we have to export this object ~~from~~ in this file using Node.js syntax.

To create this configuration, we need to know that in webpack there are four core concepts: entry point, output, loaders and plugins.

webpack.config.js:

```
const path = require('path');

module.exports = {
    entry: './src/js/index.js',
    output: {
        path: path.resolve(__dirname, 'dist/js'),
        filename: 'bundle.js'
    },
```

mode: 'development' } paste it in package.json.

};

In entry point, we specify the entry property in the object. The entry point is where webpack will start the bundling. So this is the file where it will start looking for all the dependencies. which it should then bundle together. We can specify one or more entry files. But here, we are specifying only one. and that is index.js. We have specified the path of index.js in the entry point.

Output property is where our bundle file is saved. So we specified an output property and we have to pass an object here. In this object, we put the path: the path to the folder and then the filename. The path needs to be an absolute path to have access. For that we use a builtin node package.

For that, we wrote this:

```
const path = require('path');
```

↙ just importing and putting it in path variable

'__dirname' is the current absolute path. We use path.resolve to join this current path with foxkify directory. We are just joining the current absolute path with the one that we want our bundle to be in. So webpack will output our file to dist/js directory with 'bundle.js' as name. In webpack 4, we have production & development modes. We are using development mode here.

Let's create a new file called test.js to check if they are bundled together. Add that file here: src/js/test.js.

test.js:

```
console.log('Imported module');
export default 23;
```

In the index.js, which is our entry point,

we have to import test.js.

index.js:

```
// Global app controller
import num from './test';

console.log(`I imported ${num} from another module!`);
```

We exported 23 in test.js. That will be saved into the 'num' variable

So to bundle these together, we need webpack. For that, we need to add an npm script in package.json.

package.json:

```
{
  ...
  "scripts": {
    "dev": "webpack"
  }
  ...
}
```

To use this, we need to install webpack CLI.

```
npm install webpack-cli --save-dev ↵
```

To create bundle.js:

```
npm run dev ↵
```

↳ this is the command from scripts.

Then bundle.js will be created.

`! + Tab` creates a HTML starter file in Emmd.

We can move mode: 'development' to package.json.

package.json:

```
"scripts": {
    "dev": "webpack --mode development",
    "build": "webpack --mode production"
},
```

Production mode compresses our code. So after development of our app, we can compile and compress it in production mode.

Run this:

```
npm run build
```

This compiles and compresses all the code into one file.

7. A Modern Setup: The Webpack Dev Server:

Webpack dev server automatically reloads when we save our code. It will bundle all our JavaScript files and reloads the app in a browser whenever we change a file.

To install webpack dev server:

```
npm install webpack-dev-server --save-dev
```

To configure dev server, we have to make some changes in webpack.config.js.

webpack.config.js:

```
const path = require('path');

module.exports = {
    entry: './src/js/index.js',
    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'js/bundle.js'
    },
    devServer: {
        contentBase: './dist'
    }
};
```

package.json:

```
{
  ⋮
  "scripts": {
      ⋮
      "start": "webpack-dev-server --mode
                development --open"
  },
```

Then run this command:

npm run start ↵

This will start a live reload server.

Even if we delete 'bundle.js' from 'dist' folder, this still works.

We also need to copy index.html into our 'dist' folder. We want to automatically copy it to the 'dist' folder and then inject the script tag into it. In Webpack, we use plugins to do that.

Plug-ins allow us to do complex processing of our input files and in this case, it is index.html. So we want to use a plug-in called html webpack plug-in. To use it, we have to install it.

In CMD:

```
npm install html-webpack-plugin --save-dev
```

We have to save it as a variable in webpack.config.js.

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
    entry ....
    :
    :

    devServer: {
        .....
    },

    plugins: [
        new HtmlWebpackPlugin({
            filename: 'index.html',
            template: './src/index.html'
        })
    ];
```

If we again run this command:

npm run start ↵

It will open the forkify app. This is because Webpack copied index.html to 'dist'. But it won't be visible because Webpack doesn't save data on the disk. It will stream them to the server..

To save it on disk, we need to run 'dev' or 'build' commands.

8. A Modern Setup: Babel:

Babel is a JavaScript compiler which Convert ES6, ES7 and modern version syntax to old versions. like ES5

To install Babel:

npm install babel-core babel-preset-env babel-loader --save-dev ↵

Now we need to know about loaders.

Loaders in Webpack allow us to import or load all kinds of files like converting Sass to CSS or ES6 to ES5.

Webpack.config.js:

```
.....

module: {
  rules: [
    {
      test: /\.js$/,   // this is regex. This selects all files that have .js extension
      exclude: /node_modules/,   // This line excludes node_modules folder.
      use: {
        loader: 'babel-loader'
      }
    }
  ]
}
};
```

Now we need Babel config file.

Create a new file called '.babelrc' in the project root folder.

There are somethings that we cannot convert because they are not present in the ES5 version of the language. So we need to pollyfill them to convert ES6 to ES5.

For example, the promise object is not present in ES5. But we can write ES 5 code that implements the promise in ES5 so that we can use it in our code. That is exactly what the polyfill does.

To install polyfill:

```
npm install babel-polyfill --save ↵
```

Now we need to add it in webpack.config.js and where we add it is in our entry point.

webpack.config.js:

```
module.exports = {
    entry: ['babel-polyfill', './src/js/index.js'],
    output: {
        .
        .
        .
};
```

.babelrc:

```
{
  "presets": [
      ["env", {
          "targets": {
              "browsers": [
                  "last 5 versions",
                  "ie >= 8"
              ]
          }
      }]
  ]
}
```

Installing Babel was a three step process.

→ First, we installed all the packages and added the rule with babel loader

→ Second, we created a config file in order to tell Babel which stuff we want to convert to ES5.

→ Third, we included a polyfill in order to not convert, but polyfill the features that we cannot convert with the babel loaders.

Finally, run this to see if it's working:

```
npm run dev ↵
```

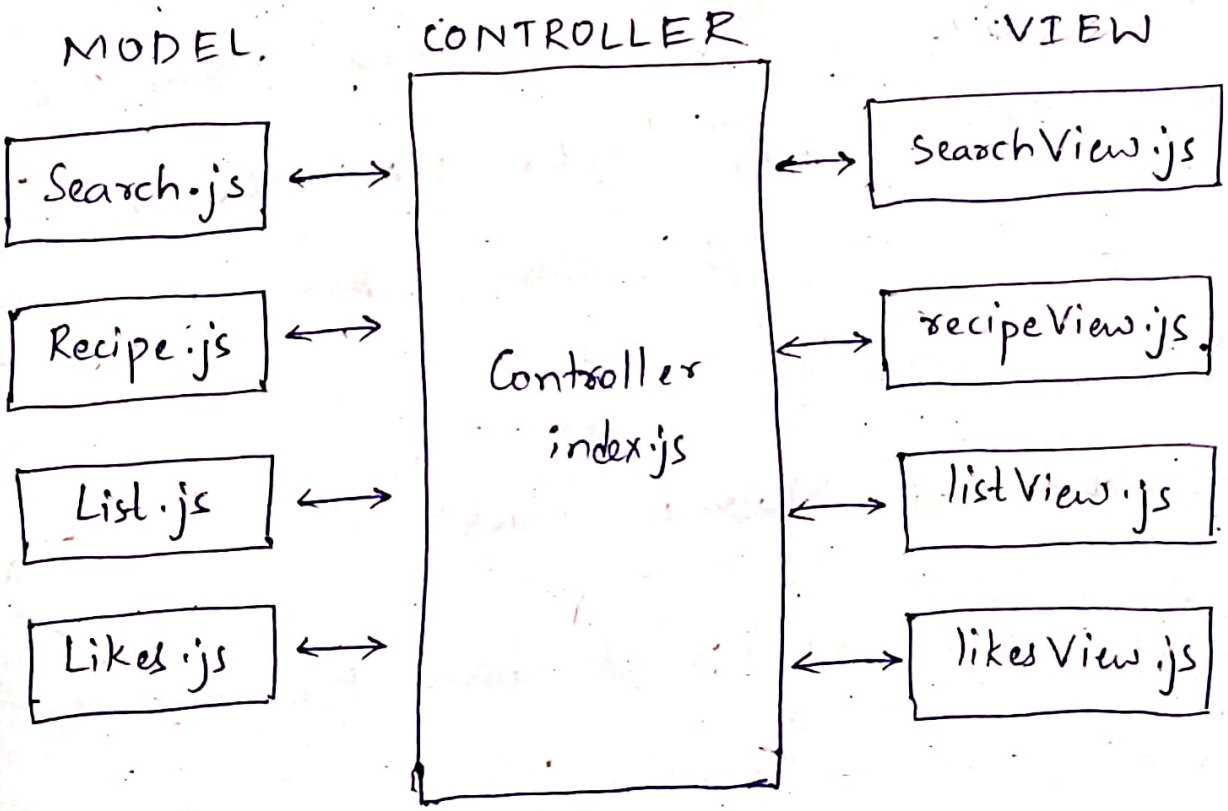
Check bundle.js for the converted code

9. Planning Our Project Architecture With MVC:

We'll use MVC architecture for this project. The main advantage of MVC architecture is that it nicely decouples

the presentation logic from application logic. with a controller in between them that controls the entire app.

This is a bit similar to what we did with the Budgety app. But here, we'll implement it with ES6 modules insted of using the module pattern that we used before. Because using real modules is a lot easier and a more real separation of concerns.



We are gonna have a controller which will have its own file called index.js. This is possible because of ES6 modules which allow us to make our JavaScript apps more modular by separating different aspects of the app into different files.

We'll have one model for each of the different aspects of the app and the same goes for views.

The Model is always concerned about the data and the app logic, while the View gets and displays data from and to the User Interface.

10. How ES6 Modules Work:

Let's have a look at how ES6 modules work.

What you will learn in this lecture:

→ How to use ES6 modules.

→ Default and named exports and imports.

We can delete test.js now. We can clear code from index.js.

Inside of src/js/models/, create a new file called Search.js. We usually write JavaScript filenames in lowercase. But in the case of models, it's a convention to use upper case letter for the model name.

Inside of src/js/views/, create a new file called searchView.js.

For these files, index.js is going to be the controller.

Default exports: We use them when we only want to export one thing from a module.

Search.js:

```
export default 'I am an exported string.';
```

We are simply exporting a string. We don't specify any variable at all. We simply put our expression so, the string right after the export and default statement.

To import this into index.js, do this:

index.js:

```
import str from './models/Search';
```

We just say import and we have to give it a name. We are calling it 'str' here. Then 'from' and the path to the module

Named Export: If we want to export multiple things from the same module, we have to use named export.

searchView.js:

```
export const add = (a,b) => a+b;
export const multiply = (a,b) => a*b;
export const ID = 23;
```

We use the export keyword but instead of default; we declare the variable that we want to export from here. Here, we are simply exporting add function and we pass two numbers into it and then we return a+b.

To import this in index.js:

index.js:

```
import str from './models/Search';
import { add, multiply, ID } from
    './views/searchView';

console.log(`Using imported functions! ${add(ID,
    2)} and ${multiply(3,5)}. ${str}`);
```

Imagine that we want to use different names ~~from~~ for the imports:

index.js:

```
import str....

import {add as a, multiply as m, ID} from
  './views/searchView';

console.log(`Using imported functions! ${a(ID,2)}
  and ${m(3,5)}. ${str}`);
```

Importing everything from a module:

index.js:

```
import str....

import * as searchView from './views/
                              searchView';

console.log(`Using imported functions! ${
  searchView.add(searchView.ID, 2)} and
${searchView.multiply(3,5)}. ${str}`);
```

Clear everything at last because its just for testing.

11. Making our First API Calls:

What you will learn in this lecture:

→ How to use a real-world API.

→ What API keys are and why we need them.

The API that we use is food2fork.com.

Go to that website → Browse → Recipe API.

This API allows us to do two types of requests. First, we can search for recipes and second, we can get data about one specific recipe.

The search request returns a list of 30 recipes and from these, we can select one of the IDs and then make a request for that specific recipe

Create a new account to get an API key.

Lets get into the code. We are not using fetch because it is a new feature which is not supported in all browsers.

So we're using a popular HTTP request library called axios. We have to install Axios first.

```
npm install axios --save
```

Axios is better than fetch because it automatically returns json but with fetch first we have to wait for the data to come back and then wait for it to convert to json. Axios is also better at error handling

index.js:

```
import axios from 'axios';

async function getResults(query) {
    const proxy = 'https://cors-anywhere.herokuapp.com/';
    const key = '....API key....';
    try {
```

```
        const res = await axios(`${proxy}http://
            food2fork.com/api/search?key=${key}&q=
            ${query}`);

        const recipes = res.data.recipes;
        console.log(recipes);
    } catch (error) {
        alert(error);
    }
}
getResults('pizza');
```

12. Building the Search - Model:

Let's now implement the MVC architecture by building model for our search functionality

What you will learn in this lecture:

→ How to build a simple data model using ES6 classes.

Search.js:

```
import axios from 'axios';

export default class Search {
    constructor(query) {
        this.query = query;
    }

    async getResults() {
        const proxy = 'https://cors-anywhere.herokuapp.com/';
        const key = '...API key?...';

        try {
            const res = await axios(`${proxy}http://food2fork.com/api/search?key=${key}&q=${this.query}`);
            this.result = res.data.recipes;
        } catch (error) {
```

```
            alert(error);
        }
    }
}
```

index.js:

```
import Search from './models/Search';

const search = new Search('pizza');
console.log(search);
search.getResults();
```

13. Building the Search Controller:

What you will learn in this lecture:

→ The concept of application state

→ A simple way of implementing state.

index.js:

```
import Search from './models/Search';
```

```
/** Global State of the app.
* - Search object
* - Current recipe object
* - Shopping list object
* - Liked recipes:
*/

const state = {}

const controlSearch = async() => {
    //1) Get query from view.
    const query = 'pizza' //TODO.
    if(query) {
        //2) New search object and add to state.
        state.search = new Search(query);

        //3) Prepare UI for results.

        //4) Search for recipes.
        await state.search.getResults();
```

```
    //5) Render results on UI
    console.log(state.search.result);
  }
}

document.querySelector('.search').addEventListener(
    'submit', e => {
  e.preventDefault();
  controlSearch();
});
```

Summary of the code:

→ We put our controller in index.js. Each model and each view will get its own file for each of the functionality. But for controllers, we're putting all of them in the same file because it makes it easier and when the file gets bigger and bigger as we are adding more controllers so it'll be easier when we have all of them in one file

→ State: Imagine our final app running with all the search queries, the recipes, the likes, the shopping list. So what is the state of our app in any given moment. Think about what is the current search query etc or whats the current recipe or how many servings are currently being calculated. So all the things in one given moment are the state. All the data in the current state in the current moment of our app, all of this data is the state and we want that to be in one central place, like one central object. For that, we wrote this:

```
const state = { }
```

All the data can be stored in one central variable which we can access throughout our controller.

→ To prevent the page from reloading when we click the search button, we wrote this:

```
document.querySel.... {
    e.preventDefault();
});
```

→ `if(query){`

```
    ⋮
}
```

This condition means:- if there is a query, get into the if block.

```
state.search = new Search(query);
```

The above line is similar to creating a new object but here we are storing it in our global state object.

→ `await state.search.getResults();`

We want the rendering of the results to happen after we receive the results from the API. So we are using await. As

we have used await, we have to say it is an async function.

→ Every asynchronous function returns a promise.

14. Building the Search View – Part 1:

What you will learn in this lecture:

→ Advanced DOM manipulation techniques.

→ How to use ES6 template strings to render entire HTML components.

→ How to create a loading spinner.

SearchView.js:

```
import { elements } from './base';

export const getInput = () => elements.
    searchInput.value;

export const clearInput = () => {
    elements.searchInput.value = '';
};
```

```
export const clearResults = () => {
    elements.searchResList.innerHTML = '';
};

const renderRecipe = recipe => {
    const markup = `
        <li>
            <a class="results__link" href="#${recipe.
            recipe_id}">
                <figure class="results__fig">
                    <img src="${recipe.image_url}"
                        alt="${recipe.title}">
                </figure>
                <div class="results__data">
                    <h4 class="results__name">${
                        recipe.title}</h4>
                    <p class="results__author">
                    ${recipe.publisher}</p>
                </div>
            </a>
        </li>
```

```
    elements.searchResList.insertAdjacentHTML('beforeend',
        markup);
};

export const renderResults = recipes => {
    recipes.forEach(renderRecipe);
};
```

index.js:

```
import Search from './models/Search';
import * as searchView from './views/searchView';
import { elements } from './views/base';

/** Global state ...
    :
*/

const state = {};

const controlSearch = async () => {
    // 1) Get query from view
```

```
const query = searchView.getInput();

if (query) {
    // 2) New search object and add to state.
    state.search = new Search(query);

    // 3) Prepare UI for results
    searchView.clearInput();
    searchView.clearResults();

    // 4) Search for recipes
    await state.search.getResults();

    // 5) Render results on UI
    searchView.renderResults(state.search.result);
  }
}

elements.searchForm.addEventListener('submit', e => {
    e.preventDefault();
    controlSearch();
});
```

base.js:

```
export const elements = {
    searchForm: document.querySelector('.search');
    searchInput: document.querySelector('.search__field
    searchResList: document.querySelector('.results__list')
};
```

Summary of the code:

→ We wrote base.js to store all our DOM elements.

→ In searchView.js:

```
export const getInput = () => elements.
                searchInput.value;
```

If we have only one line in an arrow function, then this is an implicit return. So we don't even need to write return. This returns whatever we input into the search field.

Then we're getting it in index.js and calling it query:

```
const query = searchView.getInput();
```

→ We wrote a private function called renderRecipe in searchView.js. We wrote this function to print one single recipe.

→ Then this code from searchView.js:

```
export const renderResults = recipes => {
    recipes.forEach(renderRecipe);
};
```

This function loops through all the 30 results and call renderRecipe function for each of them.

→ We use clearInput function in searchView.js to clear the search field after we search.

→ We use clearResults to clear the previous results when we search for another item.

15: Building the Search View - Part 2:

We have to reduce the title size whenever it occupies more than one line. We want entire words followed by three dots.

SearchView.js:

```
.....

export const clearResults = () => {
    elements.....
};

/*
// 'Pasta with tomato and Spinach'
acc: 0 / acc + curr.length = 5 / newTitle = ['Pasta']
```

← if we use split method on this string, it turns into an array with five elements

The accumulator in the first one is zero. Then the accumulator plus the current length will be 0 plus ~~size~~ length of 'pasta' i.e., 5. And 5 is under the limit of 17, so we can push it into the array.

acc: 5 / acc + cur.length = 9 / newTitle = ['Pasta', 'with']

acc: 9 / acc + cur.length = 15 / newTitle = ['Pasta', 'with', 'tomato']

acc: 15 / acc + cur.length = 18 / newTitle = ['Pasta', 'with', 'tomato'].

acc: 18 / acc + cur.length = 24 / newTitle = ['Pasta', 'with', 'tomato'].

*/.

```
const limitRecipeTitle = (title, limit = 17) => {
  const newTitle = [];
  if (title.length > limit) {
    title.split(' ').reduce((acc, cur) => {
      if (acc + cur.length <= limit) {
        newTitle.push(cur);
      }
      return acc + cur.length;
    }, 0);
```

reduce has accumulator and current value as input.

↳ initial value of the accumulator. Throughout the loops, we add to it

```
        // return the result
        return `${newTitle.join(' ')} ...`;
    }
    return title;
}

const renderRecipe = recipe => {
    const markup = `
        <li>
            ...
            <h4 class="...">${limitRecipeTitle(recipe.title)}</h4>
            ...
        </li>
    `;
    ...
```

16. Rendering an AJAX Loading Spinner:

We have the loader in multiple places. So we are writing it in base.js.

base.js:

```
export const elements = {
    searchForm: document.querySelector('.search'),
    searchInput: document.querySelector('.search__field'),
    searchRes: document.querySelector('.results'),
    searchResList: document.querySelector('.results__list')
};

export const elementStrings = {
    loader: 'loader'
};

export const renderLoader = parent => {
    const loader = `
        <div class="${elementStrings.loader}">
            <svg>
                <use href="img/icons.svg#icon-cw"></use>
            </svg>
        </div>
    `;
    parent.insertAdjacentHTML('afterbegin', loader);
};
```

```
export const clearLoader = () => {
    const loader = document.querySelector(`
        .${elementStrings.loader}`);
    if (loader) loader.parentElement.removeChild
                        (loader);
};
```

index.js:

```
import Search from './models/Search';
import * as searchView from './views/searchView';
import { elements, renderLoader, clearLoader }
    from './views/base';

.....

if (query) {
    .....
    // 3) Prepare UI for results
    .....
    renderLoader(elements.searchRes);
    .....
```

```
//5) Render results on UI
    clearLoader();
    searchView.renderResults(state.search.result);
}
}
```

17. Implementing Search Results Pagination:

What you will learn in this lecture:

→ How to use the '.closest' method for easier event handling.

→ How and why to use 'data-*' attributes in HTML5.

SearchView.js:

```
import { elements } from './base';
export const .....
:
export const clearResults = () => {
    elements.searchResList.innerHTML = '';
    elements.searchResPages.innerHTML = '';
```

```
};

/*
// 'Pasta with....
.
.
.

    elements.searchResList.insertAdjacentHTML('beforeend',
        markup);
};

// type: 'prev' or 'next'
const createButton = (page, type) =>
    <button class="btn-inline results__btn--${type}"
        data-goto=${type === 'prev' ? page - 1 : page + 1}
        <span>Page ${type === 'prev' ? page - 1 : page + 1}
                                                </span>
        <svg class="search__icon">
            <use href="img/icons.svg#icon-triangle-${
                type === 'prev' ? 'left' : 'right'}"></use>
        </svg>
    </button>
`;
```

```
const renderButtons = (page, numResults, resPerPage) => {
    const pages = Math.ceil(numResults / resPerPage);

    let button;
    if (page === 1 && page > 1) {
        //Only button to go to next page
        button = createButton(page, 'next');
    } else if (page < pages) {
        //Both buttons
        button = `
            ${createButton(page, 'prev')}
            ${createButton(page, 'next')}
        `;
    } else if (page === pages && pages > 1) {
        //Only button to go to prev page
        button = createButton(page, 'prev');
    }

    elements.searchResPages.insertAdjacentHTML('afterbegin',
                                                button);

};
```

```
export const renderResults = (recipes, page = 1,
    resPerPage = 10) => {
    // render results of current page
    const start = (page - 1) * resPerPage;
    const end = page * resPerPage;

    recipes.slice(start, end).forEach(renderRecipe);

    // Render pagination buttons.
    renderButtons(page, recipes.length, resPerPage);
};
```

index.js:

```
.....

elements.searchResPages.addEventListener('click', e => {
    const btn = e.target.closest('.btn-inline');
    if (btn) {
        const goToPage = parseInt(btn.dataset.goto, 10)
        searchView.clearResults();
        searchView.renderResults(state.search.result, goToPag
    }
});
```

Summary of the code:

→ We are implementing the pagination functio-nality in three steps: The first step is to change the render results function, because that's the function that will be called whenever we click on one of the buttons. So we not only need to pass in the recipes now, but we also need to pass in the page that we want to display.

The second step is to render these buttons on the interface.

The third step will be to attach event handlers to these buttons.

→ This line:

```
Const goToPage = parseInt(btn.dataset.goto,10);
console.log(goToPage);
}
```

Here, 10 means we're on base 10 i.e., from zero to nine.

18. Building the Recipe Model - Part 1:

Now that the search functionality is fully implemented, let's start working on the model for a single recipe selected from the search results.

Lets create a new file in models folder called Recipe.js.

Recipe.js:

```
import axios from 'axios';
import { key, proxy } from '../config';

export default class Recipe {
    constructor(id) {
        this.id = id;
    }

    async getRecipe() {
        try {
            const res = await axios(`${proxy}http://
                food2fork.com/api/get?key=${key}&rId=${this.id}`);
```

```
        this.title = res.data.recipe.title;
        this.author = res.data.recipe.publisher;
        this.img = res.data.recipe.image_url;
        this.url = res.data.recipe.source_url;
        this.ingredients = res.data.recipe.ingredients;
    } catch(error) {
        console.log(error);
        alert('Something went wrong :(');
    }
}

calcTime() {
    //Assuming that we need 15 min for each
      3 ingredients
    const numIng = this.indredients.length;
    const periods = Math.ceil(numIng / 3);
    this.time = periods * 15;
}
```

```
    calcServings() {
        this.servings = 4;
    }
}
```

Lets create a file called config.js to store proxy url and API key. Create it in js folder of src. We can remove them from Search.js.

config.js:

```
export const proxy = 'https://cors-anywhere.herokuapp.com/';
export const key = '462b1.... API key...';
```

index.js:

```
import Search from ....
import Recipe from './models/Recipe';
import * as ....
impo....
/** Global state....
    :
```

```
*/

Const   state  = { };

/**

*   SEARCH  CONTROLLER

*/

Const     controlSearch = ....

                .
                .
                .
                .

      SearchView.renderResults (state.search.result,
                                goToPage );

   }

});

/**

* RECIPE   CONTROLLER

*/

Const   r = new  Recipe (46956);

r.getRecipe( );
Console.log(r);
```

Recipe id from the href that we put in searchView.js.

19. Building the Recipe Controller:

In this, we will implement the controller for the recipe section.

What you will learn from this lecture:

→ How to read data from the page URL.

→ How to respond to the 'hashchange' event.

→ How to add the same event listeners to multiple events.

index.js:

```
.....

/**
* RECIPE CONTROLLER
*/
const controlRecipe = async () => {
    // Get ID from url.
    const id = window.location.hash.replace('#', '');
    console.log(id);
```

```
if (id) {

    // Prepare UI for changes.

    // Create new recipe object
    state.recipe = new Recipe(id);

    try {
        // Get recipe data
        await state.recipe.getRecipe();

        // Calculate servings and time
        state.recipe.calcTime();
        state.recipe.calcServings();

        // Render recipe
        console.log(state.recipe);

    } catch (err) {
        alert('Error processing recipe!');
    }
}
```

```
['hashchange', 'load'].forEach(event => window.
    addEventListener(event, controlRecipe));
```

We have to add a try-catch block to search Controller of index.js.

```
:
/**
* SEARCH CONTROLLER
*/
Const controlSearch.....
    :
    if (query) {
        :
        renderLoader(elements.searchRes);
        try {
            // 4) Search for recipes
            await state.search.getResults();
            // 5) Render results on UI
```

```
        clearLoader();
        searchView.renderResults(state.search.result
    } catch (err) {
        alert('Something wrong with the
                search....');
        clearLoader();
    }
}
}
```

20. Building the Recipe Model - Part 2:

Let's keep working on our recipe model now writing a complex function to process the ingredients list.

What you will learn in this lecture:

→ Use array methods like map, slice, findIndex, includes.

→ How and why to use eval.

What we'll do is to read through a list of ingredients, and in each ingredient, separate the quanility, the unit and the description of each ingredient.

·Recipe.js:

```
.....

calcServings() {
    this.servings = 4;
}

parseIngredients() {
    const unitsLong = ['tablespoons', 'tablespoon', 'ounces',
    'ounce', 'teaspoons', 'teaspoon', 'cups', 'pounds'];

    const unitsShort = ['tbsp', 'tbsp', 'oz', 'oz', 'tsp',
    'tsp', 'cup', 'pound'];

    const newIngredients = this.ingredients.map(el=>{
        //1) Uniform units
        let ingredient = el.toLowerCase();
```

```
unitsLong.forEach((unit, i) => {
    ingredient = ingredient.replace(unit,
                                    unitsShort[i]);
});

//2) Remove parentheses
ingredient = ingredient.replace(
        / *\([^)]*\) */g, ' ');

//3) Parse ingredients into count, unit
    and ingredient.
const arrIng = ingredient.split(' ');
const unitIndex = arrIng.findIndex(el2 =>
        unitsShort.includes(el2));

let objIng;
if (unitIndex > -1) {
    //There is a unit
    //Ex. 4 1/2 cups, arrCount is [4, 1/2]
        -> eval("4+1/2") -> 4.5
    //Ex. 4 cups, arrCount is [4]
```

```
const arrCount = arrIng.slice(0, unitIndex);

let count;
if (arrCount.length === 1) {
  count = eval(arrIng[0].replace('-','+'));
} else {
  count = eval(arrIng.slice(0,
    unitIndex).join('+'));
}

objIng = {
  count,
  unit: arrIng[unitIndex],
  ingredient: arrIng.slice(unitIndex+1).
    join(' ')
};
} else if (parseInt(arrIng[0], 10)){
  //There is NO unit, but 1st element is
    number
  objIng = {
```

```
            count: parseInt(arrIng[0], 10),
            unit: '',
            ingredient: arrIng.slice(1).join(' ')
        } else if (unitIndex === -1) {
            //There is NO unit and NO number
            in 1st position
            objIng = {
                count: 1,
                unit: '',
                ingredient
            }
        }

        return objIng;

    });

    this.ingredients = newIngredients;

    }

}
```

index.js:

```
...
i

/**
* RECIPE CONTROLLER
*/
const controlRecipe = async () => {
    ...
    try {
        // Get recipe data and parse ingredients.
        await state.recipe.getRecipe();
        state.recipe.parseIngredients();
        ...
```

21. Building the Recipe View - Part 1:

Let's render the recipe in the UI.

Create recipeView.js in views folder and import it in index.js.

recipeView.js:

```
import { elements } from './base';

export const clearRecipe = () => {
    elements.recipe.innerHTML = '';
};

const createIngredient = ingredient =>
    <li class="recipe__item">
        <svg class="recipe__icon">
            <use href="img/icons.svg#icon-check"></use
        </svg>
        <div class="recipe__count">${ingredient.count}
            </div>
        <div class="recipe__ingredient">
            <span class="recipe__unit">${ingredient.
                unit}</span>
            ${ingredient.ingredient}
        </div>
    </li>
`;
```

```
export const renderRecipe = recipe => {

  const markup = `

    <figure class="recipe__fig">
      <img src="${recipe.img}" alt="${recipe.
        title}" class="recipe__img">
      <h1 class="recipe__title">
        <span> ${recipe.title} </span>
      </h1>
    </figure>

    <div class="recipe__details">
      <div class="recipe__info">
        <svg class="recipe__info-icon">
          <use href="img/icons.svg#icon-stopwatch"
            </use>
        </svg>
        <span class="recipe__info-data recipe__info-
          data--minutes"> ${recipe.time} </span>
        <span class="recipe__info-text"> minutes
            </span>
      </div>
```

```
<div class="recipe__info">
  <svg class="recipe__info-icon">
    <use href="img/icons.svg#icon-man"></use>
  </svg>
  <span class="recipe__info-data recipe__info-
    data--people">${recipe.servings}</span>
  <span class="recipe__info-text">servings
  </span>

  <div class="recipe__info-buttons">
    <button class="btn-tiny"> btn-decrease
      <svg>
        <use href="img/icons.svg#icon-
          circle-with-minus"></use>
      </svg>
    </button>
    <button class="btn-tiny"> btn-increase
      <svg>
        <use href="img/icons.svg#icon-
          circle-with-plus"></use>
      </svg>
    </button>
  </div>
```

```
</div>

<button class="recipe__love">
  <svg class="header__likes">
    <use href="img/icons.svg#icon-heart-outlined">
    </use>
  </svg>
</button>

</div>

<div class="recipe__ingredients">
  <ul class="recipe__ingredient-list">
    ${recipe.ingredients.map(el => createIngredient(el))
      .join('')}
  </ul>

  <button class="btn-small recipe__btn">
    <svg class="search__icon">
      <use href="img/icons.svg#icon-shopping-cart">
      </use>
    </svg>
    <span>Add to shopping list</span>
  </button>
</div>
```

```
<div class="recipe__directions">
    <h2 class="heading-2">How to cook it</h2>
    <p class="recipe__directions-text">
        This recipe was carefully designed and
        tested by
        <span class="recipe__by">${recipe.author}
        </span>. Please checkout directions
        at their website.
    .</p>
    <a class="btn-small recipe__btn" href="
        ${recipe.url}" target="_blank">
        <span>Directions</span>
        <svg class="search__icon">
            <use href="img/icons.svg#icon-triangle-
                right"></use>
        </svg>
    </a>
</div>
`;
elements.recipe.insertAdjacentHTML('afterbegin', markup
};
```

index.js:

```
...

/**
*RECIPE   CONTROLLER.
*/

const   controlRecipe  =  async ( ) => {
    //Get  ID   from   url
     ...

    if(id) {
       //Prepare  UI  for  changes
       recipeView.clearRecipe();
       renderLoader(elements.recipe);

       //Create  new  recipe  object
        ...

       try {
          ...

          //Render  recipe
          clearLoader( );
          recipeView.renderRecipe(state.recipe);
    };
    ...
```

Recipe.js:

```
parseIngredients() {
    const units...
    const unitsShort ....
    const units = [... unitsShort, 'kg', 'g'];

    const newIngredients = this.ingredients.map(el=>{
        :
        // 3) Parse ingredients into count, unit and
           ingredient
        const arrIng = ingredient.split(' ');
        Const unitIndex = arrIng.findIndex(el2 =>
            units.includes(el2));
        :
```

22. Building the Recipe View - Part 2:

We now want to create a function which converts decimal numbers to fractions. For this, we're going to use a library called fraction.js

This library allows us to put in a number and then get a numerator and a denominator out of that number. If we input 0.75, it gives us a numerator of 3 & denominator of 4 : i.e, three quarters.

Lets install it:

```
npm install fractional --save
```

recipeView.js:

```
import { elements }....
import { Fraction } from 'fractional';

export const....
    :
};

const formatCount = count => {
    if (count) {
        // count = 2.5 --> 5/2 --> 2 1/2
        // count = 0.5 --> 1/2
        const [int, dec] = count.toString().split('.').
            map(el => parseInt(el, 10));
```

```
    if (!dec) return count;

    if (int === 0) {
        const fr = new fraction(count);
        return `${fr.numerator}/${fr.denominator
    } else {
        const fr = new Fraction(count. - int);
        return `${int} ${fr.numerator}/${
                fr.denominator}`;
    }
}
return '?';
};

Const createIngre....
    :
    :

index.js:
    :
    :
Const controlRecipe = ....
    :
    :
```

```
if (id) {

    //Prepare UI ...

    renderLoader(elements.recipe);

    // Highlight selected search item
    if (state.search) searchView.highlightSelected(id

    // Create ...
```

searchView.js:

```
import { elements } from './base';

export const ....

export const clearInput = () => {
    ....
};

export const clearResults = ...
    ....
};

export const highlightSelected = id => {
```

```
const resultsArr = Array.from(document.
    querySelectorAll('.results__link'));
resultsArr.forEach(el => {
    el.classList.remove('results__link--active');
});
document.querySelector(`a[href*="${id}"]`).
    classList.add('results__link--active');
});
```

⋮

23. Updating Recipe Servings:

What you will learn in this lecture:

→ Yet another way of implementing event delegation: ".matches".

We have to decrease or increase both the servings and the ingredients whenever we hit decrease/increase buttons.

Recipe.js:

```
    this.ingredients = newIngredients;
}

updateServings (type) {
    // Servings
    const newServings = type === 'dec' ? this.
        servings - 1 : this.servings + 1;

    // Ingredients
    this.ingredients.forEach(ing => {
        ing.count *= (newServings / this.servings)
    });

    this.servings = newServings;
}
}
```

index.js:

```
['hashchange', 'load'].forEach(event => window.
    addEventListener(event, controlRecipe));

// Handling recipe button clicks
elements.recipe.addEventListener('click', e => {
  if(e.target.matches('.btn-decrease, .btn-increase *
      )) {
    // Decrease button is clicked
    if (state.recipe.servings > 1) {
      state.recipe.updateServings('dec');
      recipeView.updateServingsIngredients(state.
        recipe);
    }
  } else if (e.target.matches('.btn-increase,
  .btn-increase *')) {
    // Increase button is clicked.
```

selects all child elements of this class.

```
        state.recipe.updateServings('inc');
        recipeView.updateServingsIngredients(state.
            recipe);
    }

    console.log(state.recipe);

});
```

recipeView.js:

```
    elements.recipe.insertAdjacentHTML('afterbegin', markup);
};

expoit const updateServingsIngredients = recipe => {
    // Update servings
    document.querySelector('.recipe__info-data--people').
        textContent = recipe.servings;

    // Update ingredients
    const countElements = Array.from(document.
        querySelectorAll('.recipe__count'));
```

```
countElements.forEach((el, i) => {
    el.textContent = formatCount(recipe.
        ingredients[i].count);
});
};
```

24. Building the Shopping List Model:

What you will learn in this lecture:

→ How and why to create unique IDs using an external package.

→ Difference between Array.slice and Array.splice.

→ More use cases for Array.findIndex and Array.find.

Now we will build our data model where we will learn how and why to create unique IDs using an external third party package.

First create a new file called List.js in src → js → models.

List.js:

npm install uniqid --save

```
import uniqid from 'uniqid';

export default class List {
    constructor() {
        this.items = [];
    }

    addItem(count, unit, ingredient) {
        const item = {
            id: uniqid(),
            count,
            unit,
            ingredient
        }
        this.items.push(item);
        return item;
    }
```

```
deleteItem(id) {
    const index = this.items.findIndex(el =>
        el.id === id);
    // [2,4,8] splice(1,2) → returns [4,8], original
        array is [2].
    // [2,4,8] slice(1,2) → returns 4, original
        array is [2,4,8]
    this.items.splice(index, 1);
}

updateCount(id, newCount) {
    this.items.find(el => el.id === id).count = newCount;
}
}
```

We are representing shopping list through an object. So we built a class which we are using as a blue print to generate the list object later in the controller.

index.js:

```
import List from './models/List';

window.l = new List();
```

25. Building The Shopping List View:

Create a new file called listView.js in src → js → views folder.

listView.js:

```
import { elements } from './base';

export const renderItem = item => {
    const markup = `
        <li class="shopping__item" data-itemid=${item.id}>
            <div class="shopping__count">
                <input type="number" value="${item.coun
                    step="${item.count}" class="shopping__
                    count-value">
                <p> ${item.unit} </p>
            </div>
```

```
        <p class="shopping__description">${item.
            ingredient}</p>
        <button class="shopping__delete btn-tiny">
            <svg>
                <use href="img/icons.svg#icon-circle-
                    with-cross"></use>
            </svg>
        </button>
    </li>
    `;
    elements.shopping.insertAdjacentHTML('beforeend',
        markup);
};

export const deleteItem = id => {
    const item = document.querySelector(`[data-
        itemid="${id}"]`);
    item.parentElement.removeChild(item);
};
```

base.js:

```
export const elements = {
    ⋮
    shopping: document.querySelector('.shopping__list')
};
⋮
```

26. Building the Shopping List Controller:

index.js:

```
import Search .....
⋮
import * as listView from './views/listView';
import { elements, ren....
⋮
const state = {};
window.state = state;
⋮
```

```
['hashchange', 'load'].forEach(event => window.
    addEventListener(event, controlRecipe));

/**
*LIST CONTROLLER
*/
Const controlList = () => {
    //Create a new list If there is none yet.
    if (!state.list) state.list = new List();

    //Add each ingredient to the list and UI
    state.recipe.ingredients.forEach(el => {
        const item = state.list.addItem(el.count,
            el.unit, el.ingredient);
        listView.renderItem(item);
    });
}
//Handle delete and update list item events
```

```
elements.shopping.addEventListener('click', e => {
    const id = e.target.closest('.shopping__item').
        dataset.itemid;

    // Handle the delete button
    if (e.target.matches('.shopping__delete, .shopping-
        delete *')) {
        // Delete from state
        state.list.deleteItem(id);

        // Delete from UI
        listView.deleteItem(id);

    // Handle the count update
    } else if (e.target.matches('.shopping__count-valu
                                ){
        const val = parseFloat(e.target.value, 10);
        state.list.updateCount(id, val);
    }
});

// Handling recipe button clicks
```

```
elements.recipe.addEventListener('click', e => {
    if (e.target.matches.....
        .
        .
        .
        recipeView.updateServingsIngredients(state.recipe
    } else if (e.target.matches('.recipe__btn--add,
        .recipe__btn--add *')) {
        controlList();
    }
});

window.l = new List();
```

ListView.js:

```
.
.
.
export const deleteItem = id => {
    const ....
    if(item) item.parentElement.removeChild(item);
};
```

27. Building the Likes Model:

Create a new file called Likes.js in src → js → models.

Likes.js:

```
export default class Likes {
    constructor() {
        this.likes = [];
    }

    addLike(id, title, author, img) {
        const like = { id, title, author, img };
        this.likes.push(like);
        return like;
    }

    deleteLike(id) {
        const index = this.likes.findIndex(el => el.id ===i
        this.likes.splice(index, 1);
    }
```

```
    isLiked (id) {
        return this.likes.findIndex (el => el.id === id) !== -1;
    }

    getNumLikes () {
        return this.likes.length;
    }
}
```

28. Building the Likes Controller:

index.js:

```
...
import Likes from './models/Likes';
...

/**
 * LIST CONTROLLER
 */
...
```

```
/**
* LIKE CONTROLLER
*/

const controlLike = () => {
    if (!state.likes) state.likes = new Likes();
    const currentID = state.recipe.id;

    // User has not yet liked current recipe
    if (!state.likes.isLiked(currentID)){
        //Add like to the state
        const newLike = state.likes.addLike(
            currentID,
            state.recipe.title,
            state.recipe.author,
            state.recipe.img
        );
        //Toggle the like button

        // Add like to UI list
        console.log(state.likes);
```

```
        //User has liked current recipe
    } else {
        //Remove like from the state
        state.likes.deleteLike(currentID);

        //Toggle the like button

        //Remove like from UI list
        console.log(state.likes);
    }
};
```

29. Building the Likes View:

Create a new file called likesView.js in src → js → views

likesView.js:

```
import { elements } from './base';
import { limitRecipeTitle } from './searchView';
```

```
export const toggleLikeBtn = isLiked => {
    const iconString = isLiked ? 'icon-heart' :
        'icon-heart-outlined';
    document.querySelector('.recipe__love use').
        setAttribute('href', `img/icons.svg#${iconString}`);
};

export const toggleLikeMenu = numLikes => {
    elements.likesMenu.style.visibility = numLikes > 0 ?
        'visible' : 'hidden';
};

export const renderLike = like => {
    const markup = `
        <li>
            <a class="likes__link" href="#${like.id}">
                <figure class="likes__fig">
                    <img src="${like.img}" alt="${like.
                        title}">
                </figure>
```

```
        <div class="likes__data">
            <h4 class="likes__name">${limitRecipeTitle(
                like.title)}</h4>
            <p class="likes__author">${like.author}
                                    </p>
        </div>
    </a>
</li>
`;
elements.likesList.insertAdjacentHTML('beforeend',
    markup);
};

export const deleteLike = id => {
    const el = document.querySelector(`.likes__
        link[href*="${id}"]`).parentElement;
    if (el) el.parentElement.removeChild(el);
}
```

index.js:

```
...

import * as likesView from './views/likesView';

...

//Create new recipe object
state.recipe = new Recipe(id);

try {
    ...

    //Render recipe
    clearLoader();
    recipeView.renderRecipe(
      state.recipe,
      state.likes.isLiked(id)
    );

}
...

//User has not yet liked current recipe
if(!.....
    ...
  state.recipe.img
```

```
        );

        // Toggle the like button
        likesView.toggleLikeBtn(true);

        // Add like to UI list
        console.log(state.likes);

    // User has liked current recipe.
    } else {

        // Remove like from the state
        state.....

        // Toggle the like button
        likesView.toggleLikedBtn(false);

        // Remove like from UI list
        console.log(state.likes);
        likesView.deleteLike(currentID);
    }

};
```

```
/**
* LIKE CONTROLLER
*/
//TESTING
state.likes = new Likes();
likesView.toggleLikeMenu(state.likes.getNumLikes());
.....
```

30. Implementing Persistent Data with localStorage:

What you will learn in this lecture:

→ How to use the localStorage API.

→ How to set, get and delete items from local storage.

In the console:

```
localStorage.setItem('id', '2hjh67')
```

Both of these should always be strings.

This sets this item into local storage.

To retrieve it:

localStorage.getItem('id') ↵

« "2hjh67"

To set another item:

localStorage.setItem('recipe', 'Tomato pasta') ↵

To view all:

localStorage ↵

▸ Storage {id: "2hjh67", loglevel: webpack-dev-server: "INFO", recipe: "Tomato pasta", length: 3}

To remove item:

localStorage.removeItem('id') ↵

index.js:

```
    ...

    likesView.toggleLikeMenu(state.likes.getNumLikes());
};

//Restore liked recipes on page load
window.addEventListener('load', () => {
```

```
state.likes = new Likes();

//Restore likes
state.likes.readStorage();

//Toggle like menu button
likesView.toggleLikeMenu(state.likes.getNumLikes());

//Render the existing likes
state.likes.likes.forEach(like => likesView.
    renderLike(like));
});
```

⋮

Likes.js:

```
export.....
```

⋮

```
addLike(id, title, ....){
    const ...
    this.likes....

    //Persist data in localStorage.
```

```
        this.persistData();

        return like;
    }

    deleteLike(id) {
        const ....
        this.likes.splice(index, 1);

        // Persist data in local storage
        this.persistData();
    }

    .....

    persistData() {
        localStorage.setItem('likes', JSON.stringify(this.
            likes));
    }

    readStorage() {
        const storage = JSON.parse(localStorage.
            getItem('likes'));
```

```
    //Restoring likes from the local storage.
    if(storage) this.likes = storage;
  }
}
```

31. Wrapping Up: Final Considerations:

We have some bugs in formatting numbers.

Let's fix them:

recipeView.js:

```
:
const formatCount = count => {
  if (count) {
    //...
    //...
    const newCount = Math.round(count * 10000)/10000;
```

In the remaining function, replace count with newCount.