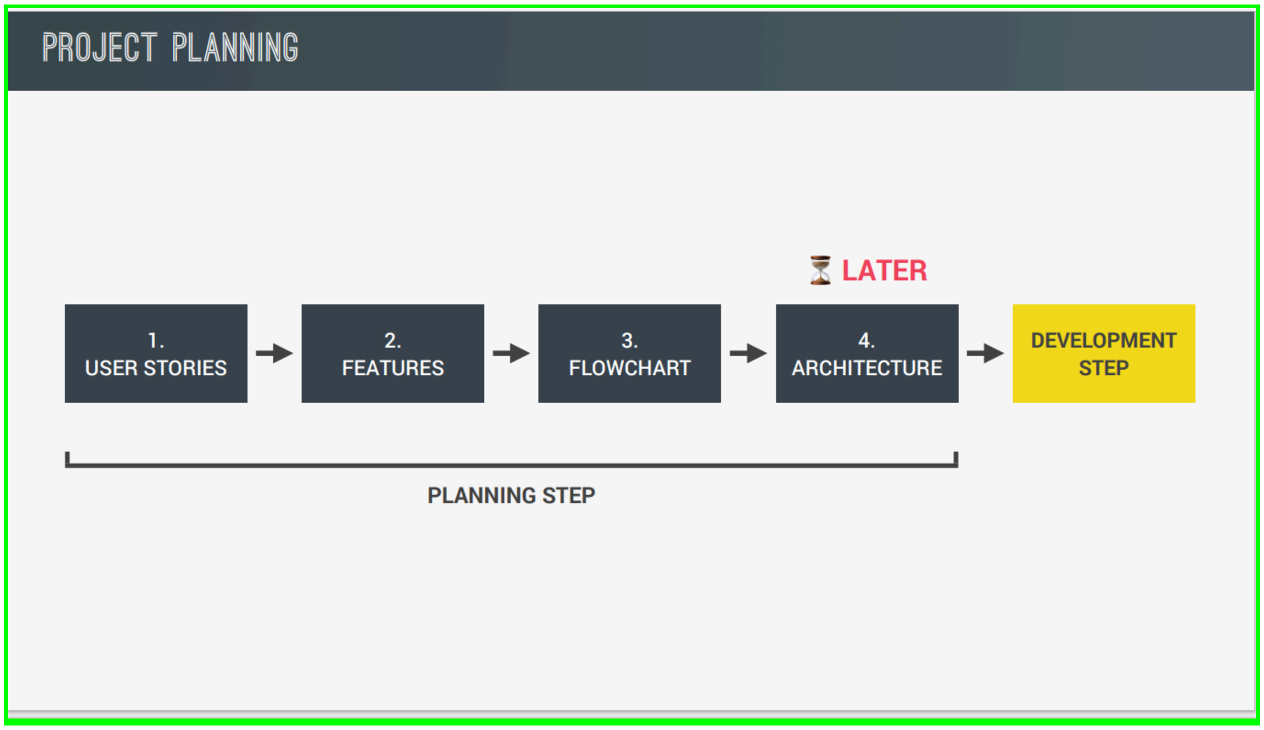
Forkify App

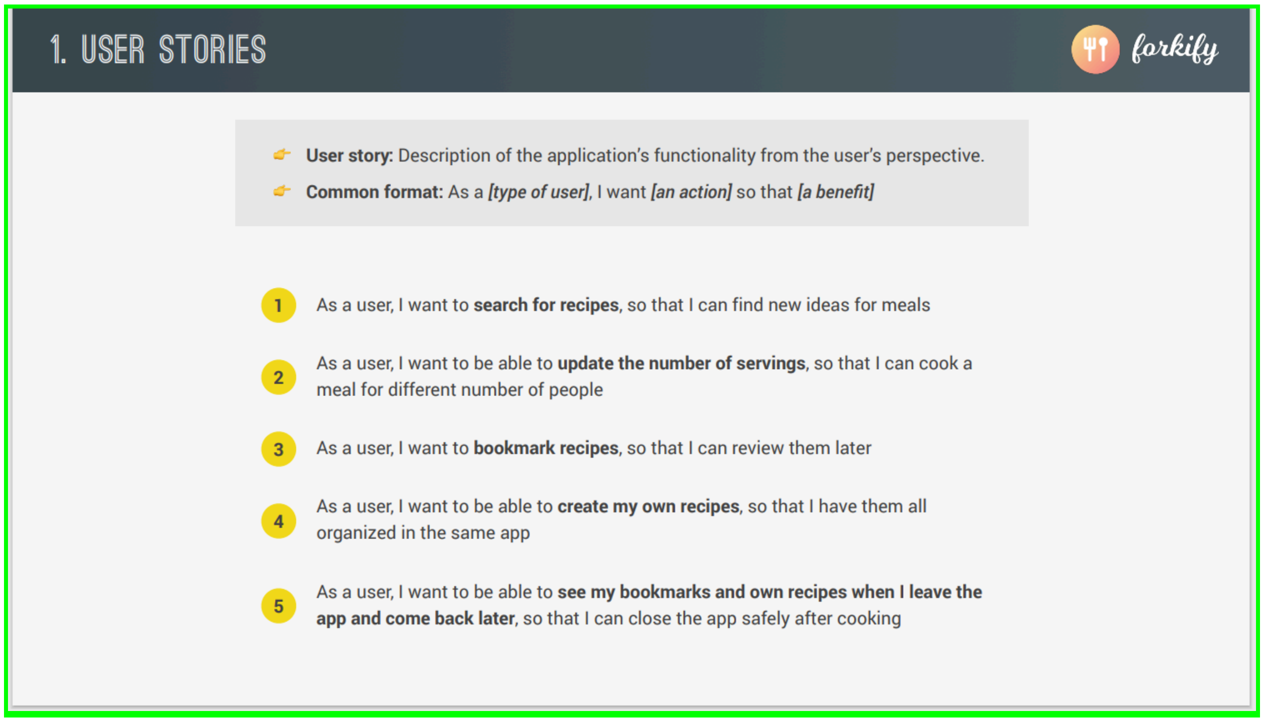


PROJECT PLANNING

⌛ LATER

1. USER STORIES → 2. FEATURES → 3. FLOWCHART → 4. ARCHITECTURE → DEVELOPMENT STEP

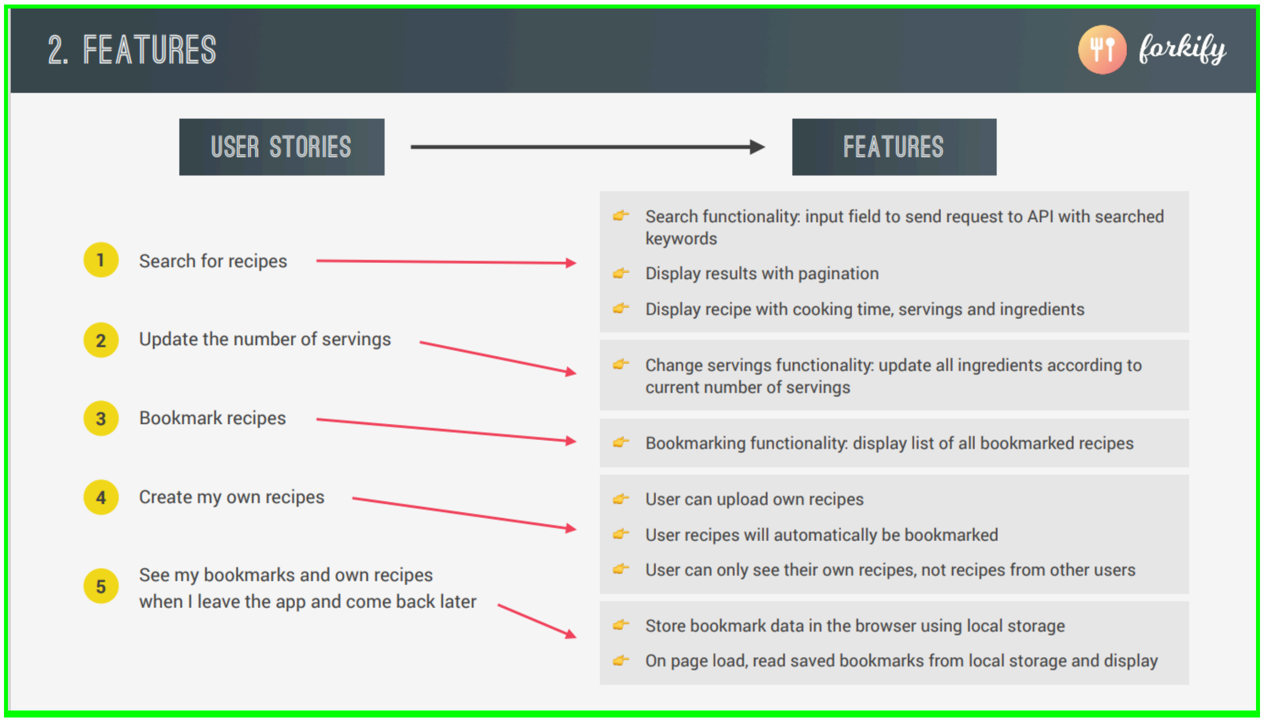
PLANNING STEP



1. USER STORIES

- **User story:** Description of the application's functionality from the user's perspective.
- **Common format:** As a *[type of user]*, I want *[an action]* so that *[a benefit]*

1. As a user, I want to **search for recipes**, so that I can find new ideas for meals
2. As a user, I want to be able to **update the number of servings**, so that I can cook a meal for different number of people
3. As a user, I want to **bookmark recipes**, so that I can review them later
4. As a user, I want to be able to **create my own recipes**, so that I have them all organized in the same app
5. As a user, I want to be able to **see my bookmarks and own recipes when I leave the app and come back later**, so that I can close the app safely after cooking

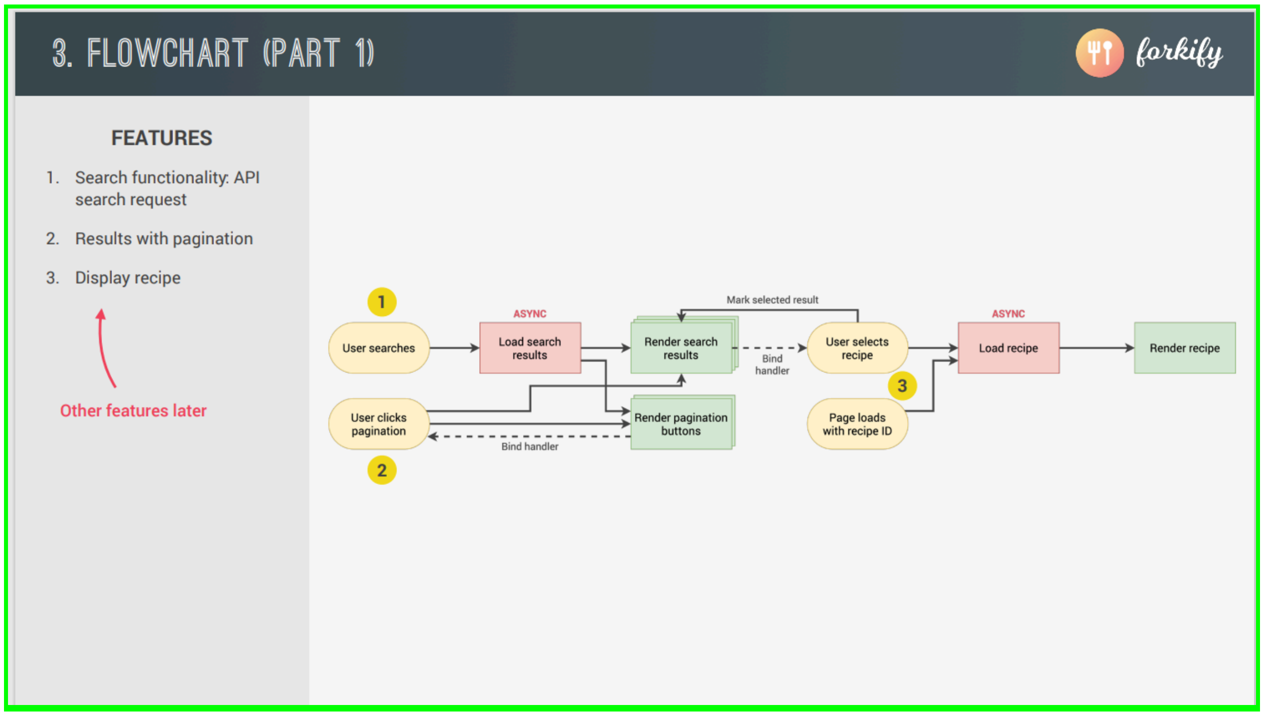


2. FEATURES

forkify

USER STORIES → FEATURES

1. Search for recipes →
   - Search functionality: input field to send request to API with searched keywords
   - Display results with pagination
   - Display recipe with cooking time, servings and ingredients
2. Update the number of servings →
   - Change servings functionality: update all ingredients according to current number of servings
3. Bookmark recipes →
   - Bookmarking functionality: display list of all bookmarked recipes
4. Create my own recipes →
   - User can upload own recipes
   - User recipes will automatically be bookmarked
   - User can only see their own recipes, not recipes from other users
5. See my bookmarks and own recipes when I leave the app and come back later →
   - Store bookmark data in the browser using local storage
   - On page load, read saved bookmarks from local storage and display



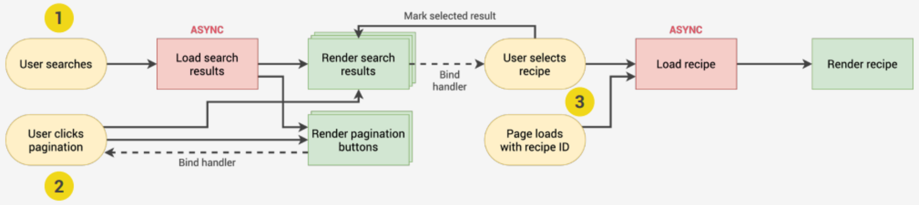
3. FLOWCHART (PART 1)

forkify

FEATURES

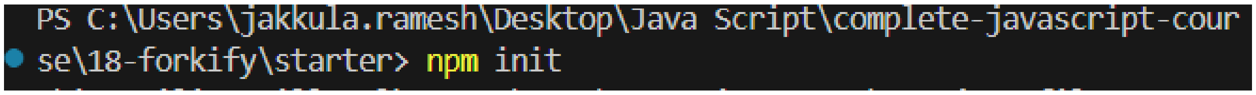
1. Search functionality: API search request
2. Results with pagination
3. Display recipe

Other features later

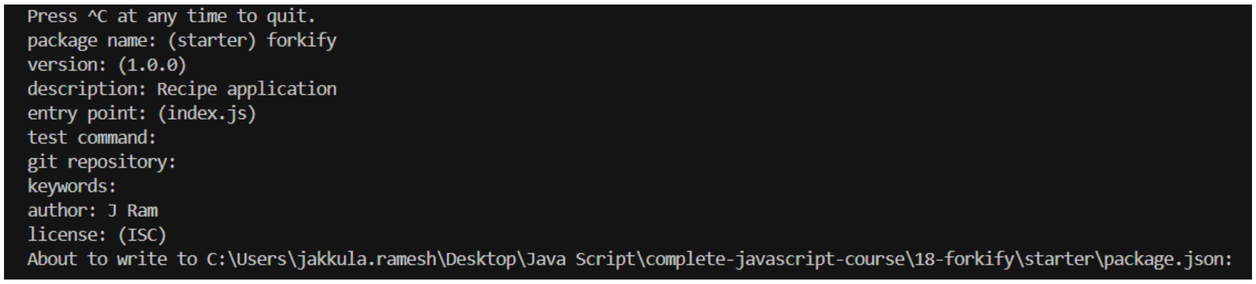


1) Initializing npm in working folder/project , then it will create package.json file

```
npm init
```

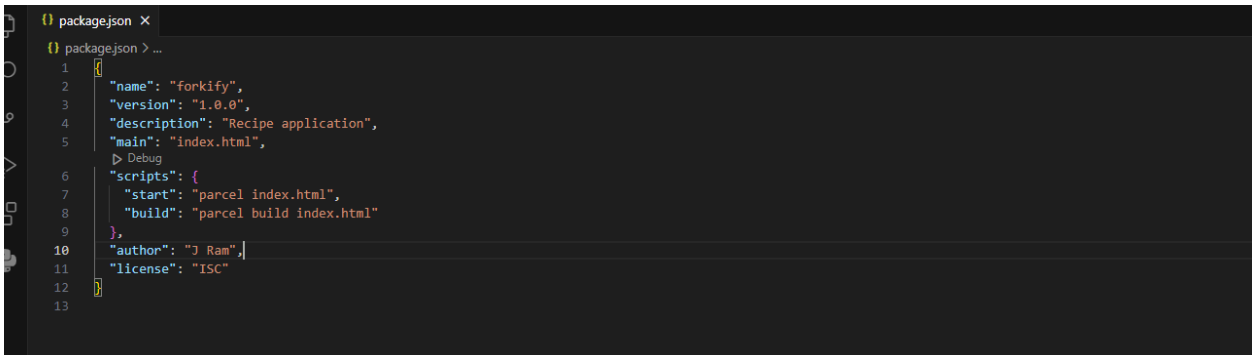


```
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-cour
se\18-forkify\starter> npm init
```



```
Press ^C at any time to quit.
package name: (starter) forkify
version: (1.0.0)
description: Recipe application
entry point: (index.js)
test command:
git repository:
keywords:
author: J Ram
license: (ISC)
About to write to C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\18-forkify\starter\package.json:
```

2)Add start and build as startup command for parcel.



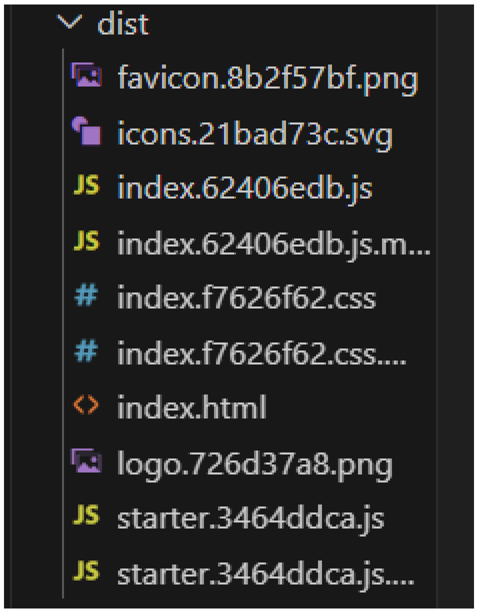
```
{
  "name": "forkify",
  "version": "1.0.0",
  "description": "Recipe application",
  "main": "index.html",
  "scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html"
  },
  "author": "J Ram",
  "license": "ISC"
}
```

3) install parcel

You can install Parcel globally or as a dev dependency in your project.

# Locally (recommended for projects)

npm install --save-dev parcel



```
dist
  favicon.8b2f57bf.png
  icons.21bad73c.svg
  index.62406edb.js
  index.62406edb.js.m...
  index.f7626f62.css
  index.f7626f62.css....
  index.html
  logo.726d37a8.png
  starter.3464ddca.js
  starter.3464ddca.js....
```

Using third-party api created by mentor

https://forkify-api.herokuapp.com/v2

```
import icons from 'url:../img/icons.svg';
```

It will point to the http://localhost:1234/icons.dfd7a6db.svg?1734944777367
Icons that were created in the **dist** folder.

Loading spinner
A spinner is a visual element used to indicate that a process (like loading data or waiting for a response) is ongoing.
can create a simple loading spinner using CSS @keyframes and the rotate animation:

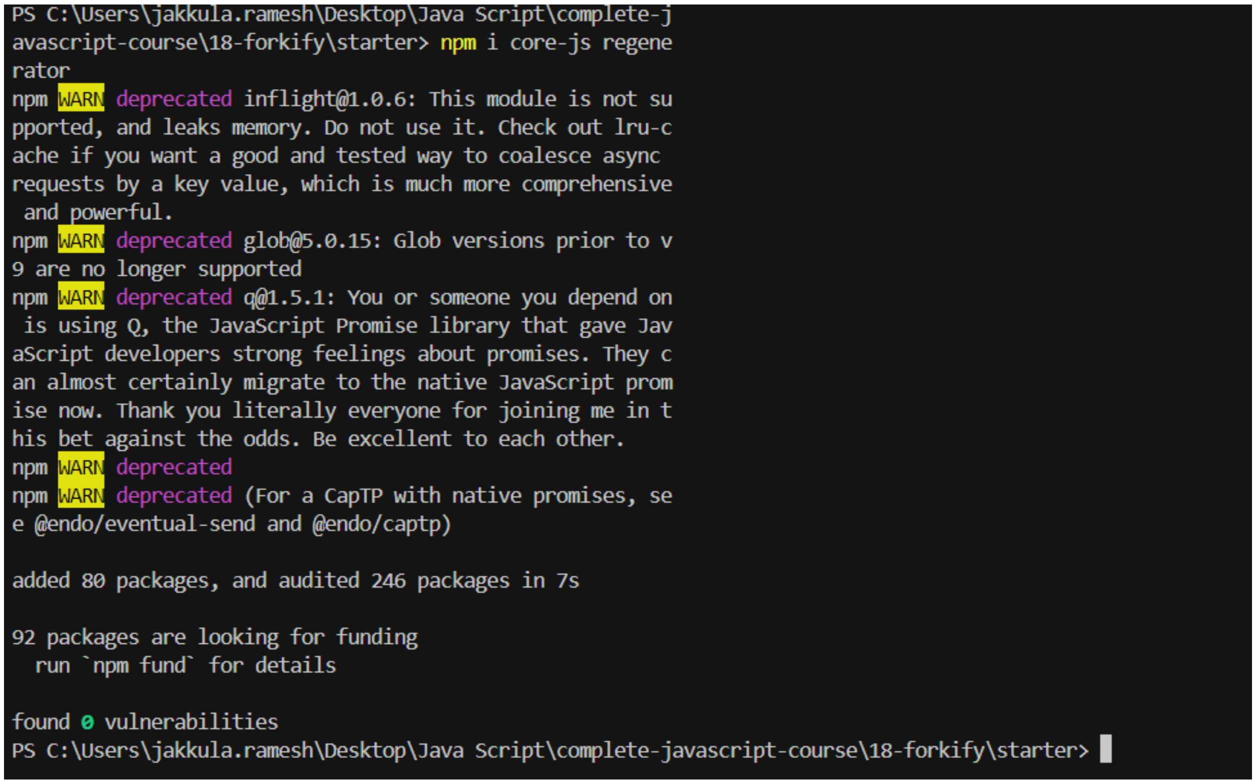
```
/* Spinner container */
.spinner {
  margin: 5rem auto;
  text-align: center;
}

/* SVG spinner animation */
.spinner svg {
  width: 6rem;
  height: 6rem;
```

```
 animation: rotate 2s infinite linear ;
}
```

```
/* Keyframes for rotation */
 @keyframes rotate {
  0% {
    transform: rotate(0deg);
  }
  100% {
    transform: rotate(360deg);
  }
```

Add polyfill features for es6 features to our code base
Install
npm i core-js regenerator (installing multiple packages at a time two packages.)

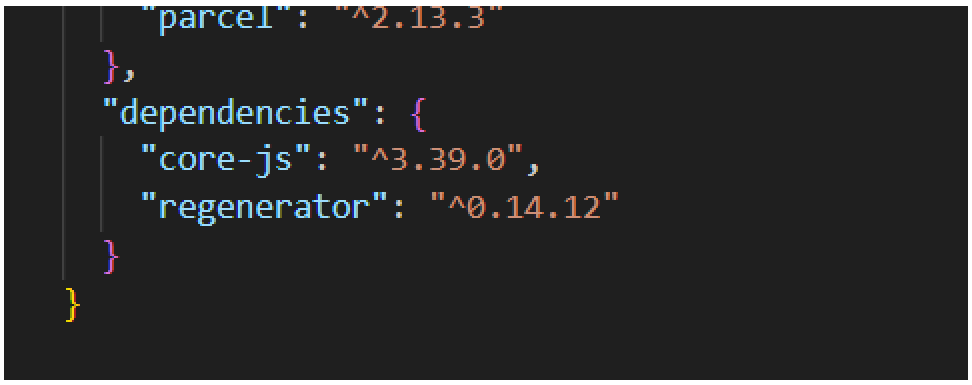


```
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-j
avascript-course\18-forkify\starter> npm i core-js regene
rator
npm WARN deprecated inflight@1.0.6: This module is not su
pported, and leaks memory. Do not use it. Check out lru-c
ache if you want a good and tested way to coalesce async
requests by a key value, which is much more comprehensive
 and powerful.
npm WARN deprecated glob@5.0.15: Glob versions prior to v
9 are no longer supported
npm WARN deprecated q@1.5.1: You or someone you depend on
 is using Q, the JavaScript Promise library that gave Jav
aScript developers strong feelings about promises. They c
an almost certainly migrate to the native JavaScript prom
ise now. Thank you literally everyone for joining me in t
his bet against the odds. Be excellent to each other.
npm WARN deprecated
npm WARN deprecated (For a CapTP with native promises, se
e @endo/eventual-send and @endo/captp)

added 80 packages, and audited 246 packages in 7s

92 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\18-forkify\starter>
```



```
    "parcel": "^2.13.3"
  },
  "dependencies": {
    "core-js": "^3.39.0",
    "regenerator": "^0.14.12"
  }
}
```

```
import 'core-js/stable';
import 'regenerator-runtime/runtime';
```

The use of the following imports in your project is to ensure **cross-browser compatibility** and **support for modern JavaScript features** in environments that don't natively support them:

```
import 'core-js/stable';
import 'regenerator-runtime/runtime';
```

Here's a detailed explanation of why these are useful:

---

1. core-js/stable

Purpose:

- core-js/stable provides **polyfills** for modern JavaScript features, such as:
  - Promises (Promise)
  - New methods for arrays (Array.prototype.includes, Array.prototype.flat)
  - New methods for objects (Object.entries, Object.values)
  - Symbols and other ES6+ features

Why Use It:

- When you write code using these modern features, they might not work in older browsers like Internet Explorer or outdated versions of Safari, Chrome, or Edge.
- core-js/stable ensures these features work by replacing them with equivalent, older implementations (polyfills).

Example:

```
// Modern JavaScript feature
const arr = [1, 2, 3];
console.log(arr.includes(2)); // true
```

Without `core-js/stable`, older browsers (e.g., IE11) would throw an error because `Array.prototype.includes` is not supported. With the polyfill, the method will work.

---

2. regenerator-runtime/runtime

Purpose:

- `regenerator-runtime/runtime` provides support for **generator functions** and **async/await** in environments that do not natively support them.

Why Use It:

- When you write asynchronous JavaScript using `async` and `await`, or generator functions (`function*`), they are typically transpiled to ES5 by tools like **Babel**.
- This transpilation relies on `regenerator-runtime` to provide the runtime support required for these features.

Example:

```
// Async function
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  return response.json();
}
```

Older environments don't support `async/await`. Babel transforms this code into something like:

```
function fetchData() {
  return regeneratorRuntime.async(function fetchData$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
```

```
        _context.next = 2;
        return fetch('https://api.example.com/data');

      case 2:
        return _context.abrupt("return", _context.sent.json());
    }
  }
 });
}
```

window.addEventListener.

The regenerator-runtime library is required to execute this transformed code.

The correct method for adding an event listener to the window object is window.addEventListener. This method listens for events of a specified type and executes a callback function when the event occurs.

Syntax

```
window.addEventListener(type, listener, options);
```

zrameters

type (string): The type of event to listen for, such as:

- 'load': Fires when the page finishes loading.
- 'resize': Fires when the window is resized.
- 'scroll': Fires when the user scrolls the window.
- 'click': Fires when the user clicks anywhere in the window.
- 'keydown': Fires when a key is pressed.

listener (function): A callback function that is executed when the event is triggered. Example:

```
function handleEvent(event) {
  console.log('Event triggered:', event);
}
```

options (optional):

- **capture**: A boolean indicating whether the event should be captured during the capture phase.

- **once**: If true, the listener is automatically removed after being called once.
- **passive**: If true, the listener will not call preventDefault(), improving performance for scroll events.

Examples

1. Listening for the Page Load Event

```
window.addEventListener('load', () => {
  console.log('Page is fully loaded!');
});
```

2. Listening for Window Resize

```
window.addEventListener('resize', () => {
  console.log('Window resized to:', window.innerWidth, 'x', window.innerHeight);
});
```

3. Listening for Scroll Event

```
window.addEventListener('scroll', () => {
  console.log('Page scrolled! Scroll position:', window.scrollY);
});
```

4. Using Options (e.g., once and passive)

```
window.addEventListener(
  'click',
  () => {
    console.log('This will only run once!');
  },
  { once: true }
);
```

Removing an Event Listener

To remove an event listener, use window.removeEventListener. The function reference must match the one used in addEventListener.

Example:

```
function handleResize() {
  console.log('Window resized!');
}

// Add event listener
window.addEventListener('resize', handleResize);

// Remove event listener
window.removeEventListener('resize', handleResize);
```

Common Use Cases

- Detecting when the user resizes the browser window (resize).
- Tracking user scroll position (scroll).
- Running code after the page has fully loaded (load).

Guard classes

A guard **class** typically refers to a class or method that is responsible for performing checks or validations to ensure certain conditions are met before an operation or action is allowed to proceed. The purpose of a guard is to protect or enforce rules, often around input, permissions, or state, to avoid errors or undesired behavior.

The concept of a **guard** is frequently seen in the following contexts:

1. Guard Clauses in Functions

A guard clause is an early exit condition in a function or method that prevents unnecessary computations if a condition is not met. It improves readability by reducing nested conditionals.

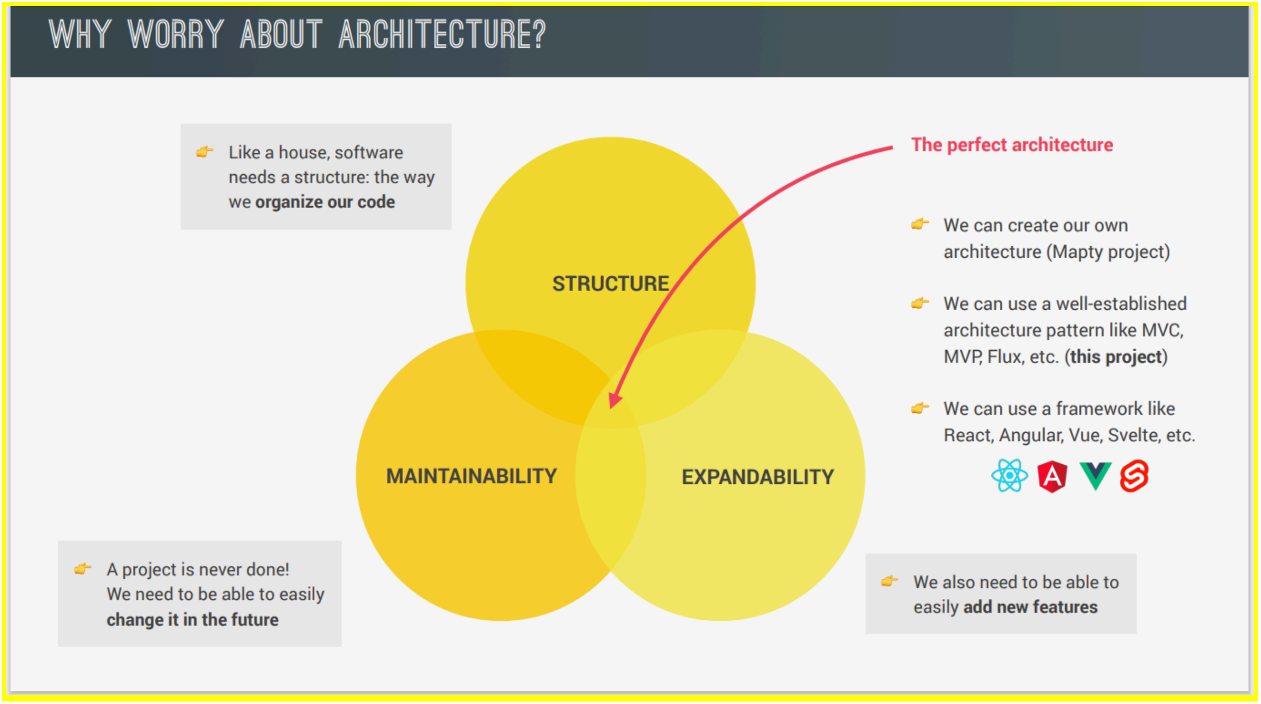
Example of Guard Clause in a Method (JavaScript):

```
function processData(data) {
  // Guard clause: if data is null or undefined, exit early
  if (!data) {
    console.log("No data provided");
    return;
  }

  // Normal processing
  console.log("Processing data:", data);
}
```

In this case, the guard clause ensures that if data is not provided or is falsy, the function will exit early, avoiding further processing.

The MVC architecture



WHY WORRY ABOUT ARCHITECTURE?

👉 Like a house, software needs a structure: the way we organize our code

👉 A project is never done! We need to be able to easily **change it in the future**

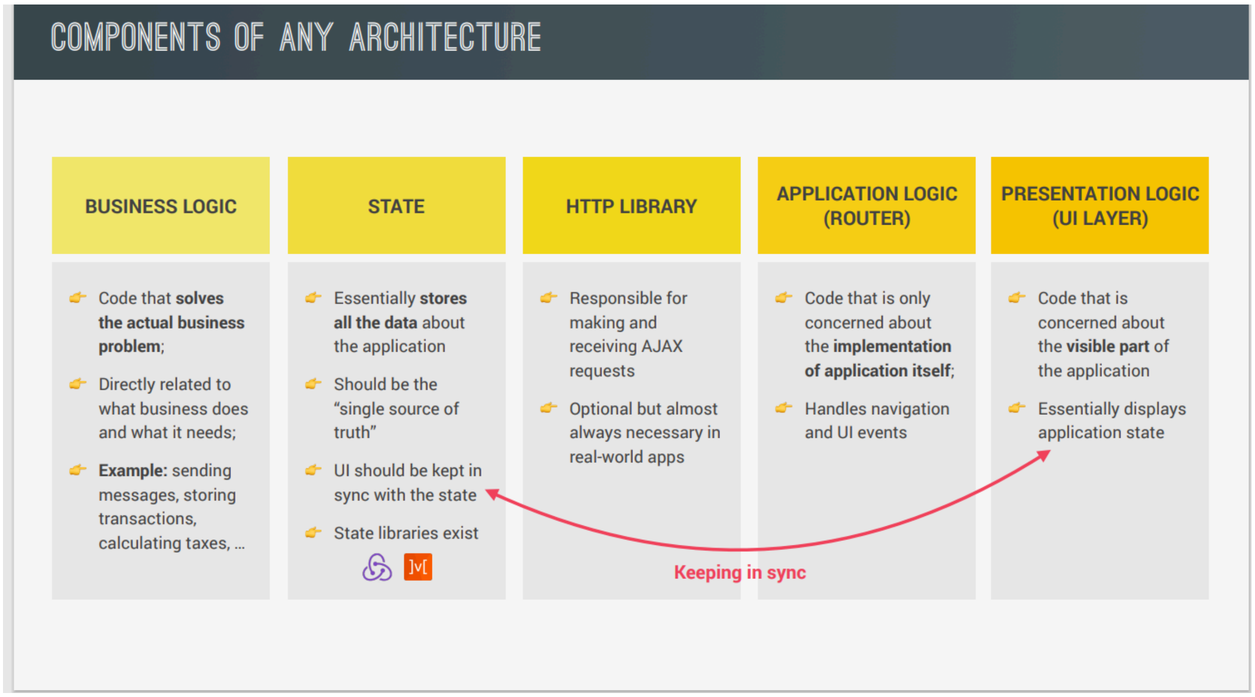
👉 We also need to be able to easily **add new features**

The perfect architecture

👉 We can create our own architecture (Mapty project)

👉 We can use a well-established architecture pattern like MVC, MVP, Flux, etc. **(this project)**

👉 We can use a framework like React, Angular, Vue, Svelte, etc.



COMPONENTS OF ANY ARCHITECTURE

BUSINESS LOGIC

- Code that **solves the actual business problem**;
- Directly related to what business does and what it needs;
- **Example:** sending messages, storing transactions, calculating taxes, …

STATE

- Essentially **stores all the data** about the application
- Should be the "single source of truth"
- UI should be kept in sync with the state
- State libraries exist

HTTP LIBRARY

- Responsible for making and receiving AJAX requests
- Optional but almost always necessary in real-world apps

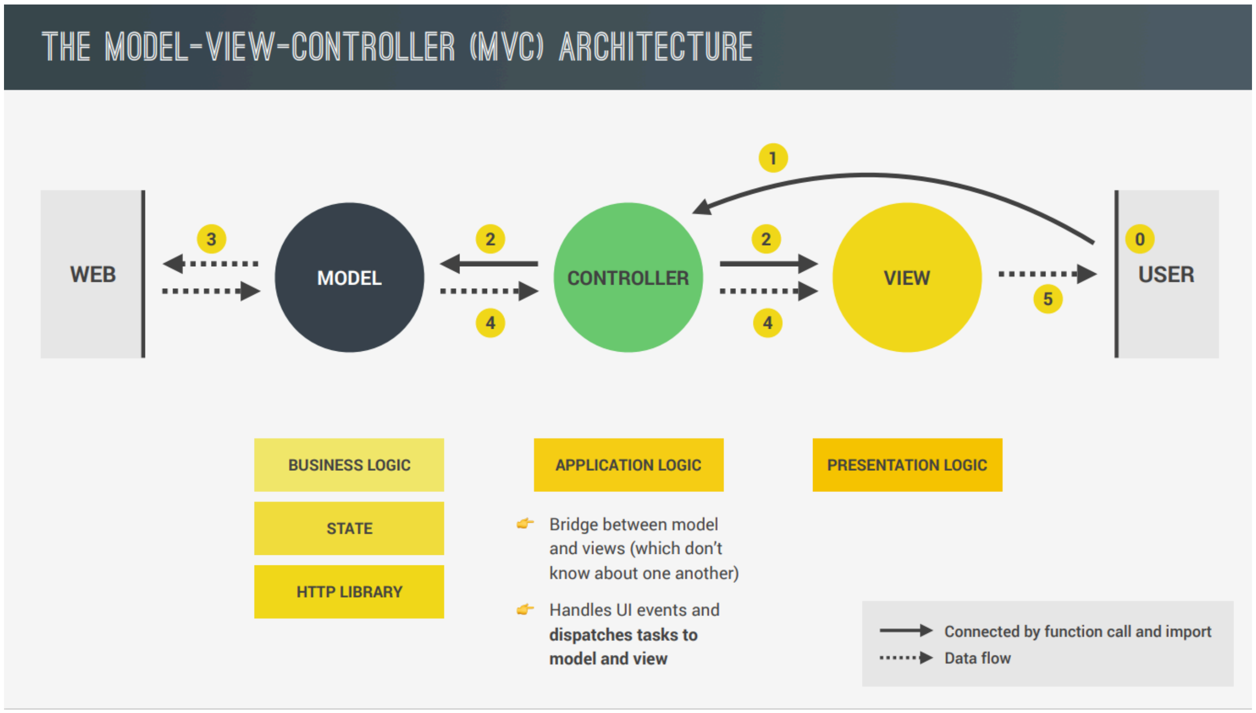
APPLICATION LOGIC (ROUTER)

- Code that is only concerned about the **implementation of application itself**;
- Handles navigation and UI events

PRESENTATION LOGIC (UI LAYER)

- Code that is concerned about the **visible part** of the application
- Essentially displays application state

Keeping in sync



THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE

BUSINESS LOGIC

STATE

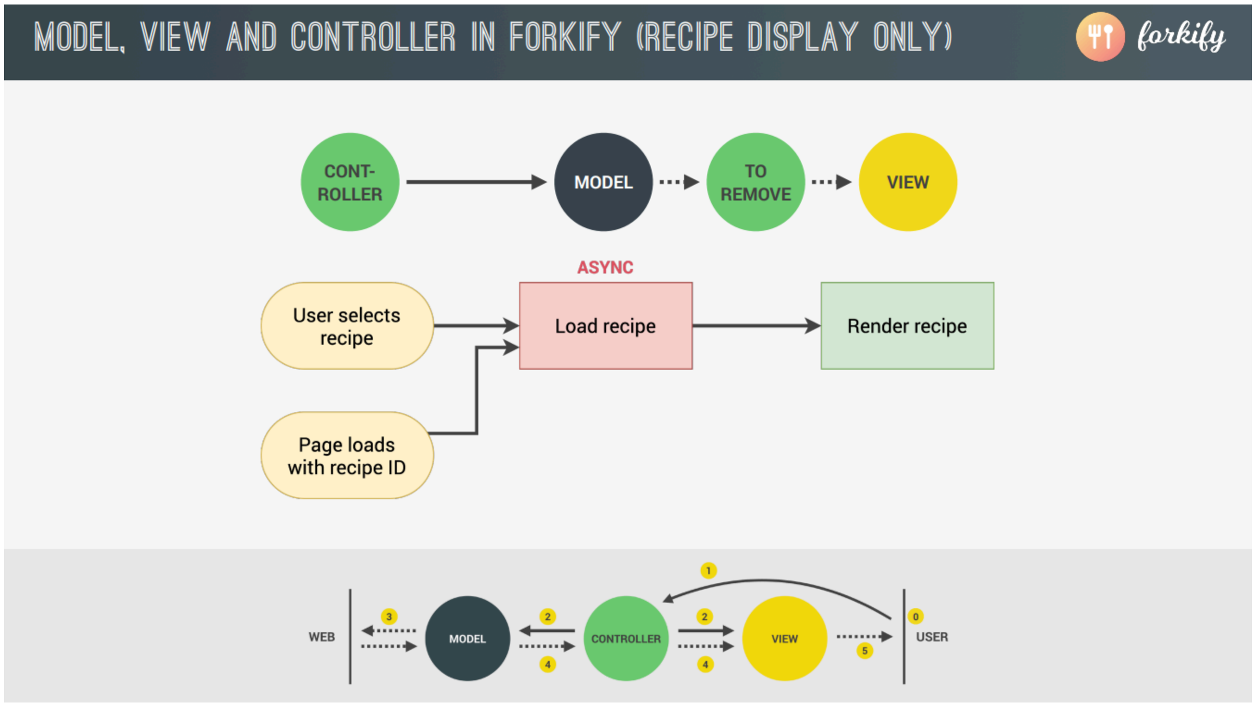
HTTP LIBRARY

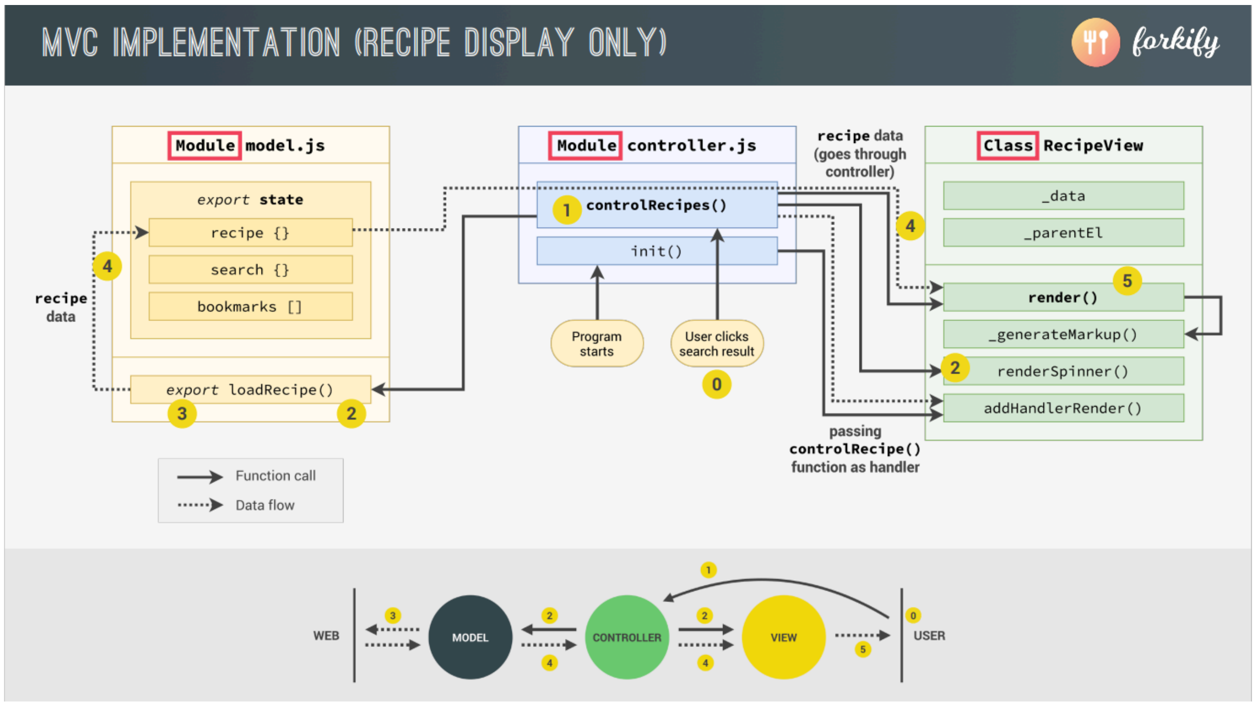
APPLICATION LOGIC

- Bridge between model and views (which don't know about one another)
- Handles UI events and **dispatches tasks to model and view**

PRESENTATION LOGIC

⟶	Connected by function call and import
⋯⟶	Data flow





The **MVC (Model-View-Controller)** architecture is a design pattern commonly used in web development to separate the concerns of an application. This separation enhances maintainability, scalability, and testability. In JavaScript, the MVC architecture typically divides the application into three main components:

Model:

- The Model represents the data and business logic of the application.
- It directly manages the data, logic, and rules of the application. This could be data fetched from a database, user input, or any other source.
- In JavaScript, the model can be represented by objects, arrays, or classes that hold data and perform necessary operations on it.

Example:

```
class Model {
   constructor() {
      this.data = [];
   }

   fetchData() {
      // Logic to fetch data
      this.data = [1, 2, 3, 4];
   }

   getData() {
      return this.data;
   }
}
```

View:

- The View is responsible for rendering the user interface (UI) and displaying data.
- It listens for updates from the Model and reflects changes to the user interface accordingly.
- In JavaScript, this can be DOM elements or frameworks/libraries like React, Vue, or Angular.

Example:

```
class View {
   constructor() {
      this.outputElement = document.getElementById('output');
   }

   render(data) {
        this.outputElement.innerHTML = data.join(', ');
    }
}
```

Controller:

- The Controller acts as an intermediary between the Model and View.
- It takes input from the user (typically via the View), processes it (with the help of the Model), and updates the View accordingly.
- In JavaScript, the controller handles events, updates models, and triggers changes in the View.

Example:

```
class Controller {
    constructor(model, view) {
        this.model = model;
        this.view = view;
    }

    updateData() {
        this.model.fetchData();
        this.view.render(this.model.getData());
    }

}
```

Implementing in project

npm install fractional

`npm install fractional` is a command used to install the **Fractional** library from the Node Package Manager (NPM). The **Fractional** library is a JavaScript utility that helps in representing and performing calculations with fractions.

What is Fractional?

The Fractional library provides a way to:

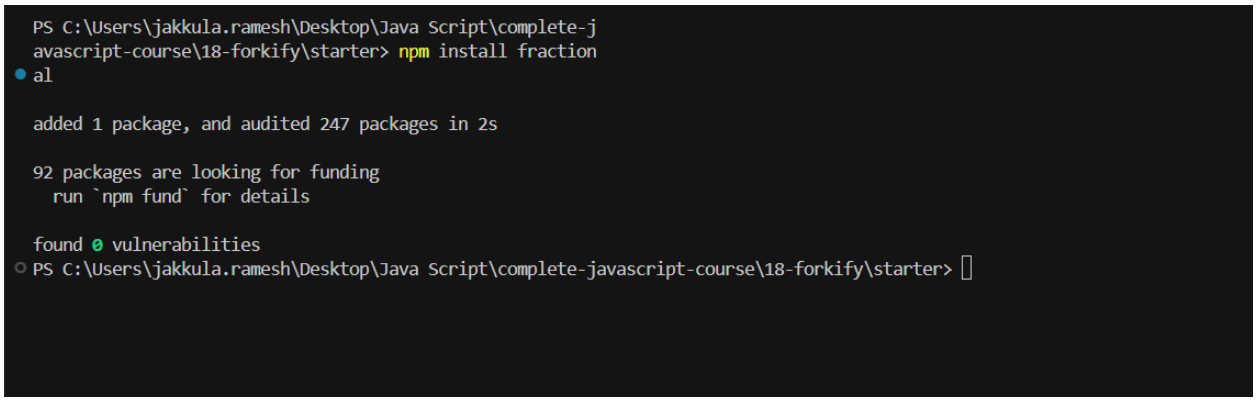
- Represent numbers as fractions (e.g., `1/2`, `3/4`).

- Perform arithmetic operations (e.g., addition, subtraction, multiplication, and division) using fractions.
- Simplify fractions to their lowest terms.

Installing the Library

To install the library, use:

```
npm install fractional
```



```
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-j
avascript-course\18-forkify\starter> npm install fraction
al

added 1 package, and audited 247 packages in 2s

92 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\18-forkify\starter>
```

Use Case

You might use **Fractional** when you need to work with fractional arithmetic or represent numerical results as fractions, rather than as decimals. This is especially useful in fields like:

- Math-based applications (e.g., calculators or educational tools).
- Financial computations where precise fractional representation is needed.
- Applications involving measurement units (e.g., recipes, construction, etc.).

Example with ES Modules (Modern JavaScript)

```
// Import the Fraction class using ES Modules
import { Fraction } from 'fractional';

// Convert decimal numbers to fractions
const frac1 = new Fraction(0.5); // 0.5 = 1/2
const frac2 = new Fraction(0.25); // 0.25 = 1/4

// Add the fractions
const sum = frac1.add(frac2); // 1/2 + 1/4 = 3/4
console.log(`Sum: ${sum.toString()}`); // Output: "Sum: 3/4"
```

```
// Subtract the fractions
const difference = frac1.subtract(frac2); // 1/2 - 1/4 = 1/4
console.log(`Difference: ${difference.toString()}`); // Output: "Difference: 1/4"

// Multiply the fractions
const product = frac1.multiply(frac2); // 1/2 * 1/4 = 1/8
console.log(`Product: ${product.toString()}`); // Output: "Product: 1/8"

// Divide the fractions
const quotient = frac1.divide(frac2); // (1/2) ÷ (1/4) = 2/1 or 2
console.log(`Quotient: ${quotient.toString()}`); // Output: "Quotient: 2"
```

Publisher-subscriber pattern

The **Publisher-Subscriber Pattern** (also known as **Pub-Sub** pattern) is a design pattern used to manage communication between components or objects. It allows one component (the *Publisher*) to send messages to multiple components (the *Subscribers*) without knowing who or what those subscribers are. This creates a decoupled architecture, where the publisher and subscriber don't need to be aware of each other directly.

Key Concepts:

- **Publisher**: The entity that publishes messages or events. It is unaware of the subscribers and doesn't know how many or who the subscribers are.
- **Subscriber**: The entity that subscribes to specific events or messages. It reacts to those events when they are triggered.
- **Event**: The message or update that the publisher sends out, and that subscribers listen for.

Real-Time Example:

Imagine a **news agency** (the Publisher) that broadcasts news updates (events) to multiple people (the Subscribers). The news agency doesn't need to know who is receiving the news or how many people are subscribed. Each person who subscribes to the news updates will receive the latest news as soon as it is published.

—---

In the code snippet provided, the **publisher-subscriber pattern** is implemented to handle changes in the browser's URL (via hashchange) or when the page loads (via load). Let's break down the relationship and its role.

Ex1:

The **Publisher-Subscriber Pattern** (Pub-Sub) is a messaging pattern that allows a **publisher** (the entity that generates data or events) to send messages to **subscribers** (entities that listen and react to those messages), without the publisher needing to know who or how many subscribers there are. This helps achieve **decoupling** in the system, making it more flexible and scalable.

Key Concepts:

- **Publisher**: This is the component that *publishes* or *sends out* events or messages. It doesn't know anything about the subscribers.
- **Subscriber**: This is the component that *subscribes* to specific events or messages and reacts when they are received. It doesn't know who the publisher is.
- **Event/Message**: The actual information or event that is sent by the publisher and received by the subscribers.

Simple Flow:

1. **Publisher** emits an event or message.
2. **Subscribers** who are listening for that event get the message and process it.

Example:

Let’s break it down with a very simple example in JavaScript:

Step-by-Step Example:

1. **Publisher** creates an event (message) and sends it out.
2. **Subscriber** listens for that message event and does something when it’s received.

```
// 1. Publisher: Sends a message to subscribers
class Publisher {
  constructor() {
    this.subscribers = [];  // List of subscribers
  }

  subscribe(subscriber) {
    this.subscribers.push(subscriber);  // Adds a new subscriber
  }

  publish(message) {
    // Sends the message to all subscribers
    this.subscribers.forEach(subscriber => subscriber.receive(message));
  }
}

// 2. Subscriber: Reacts to messages
class Subscriber {
  constructor(name) {
    this.name = name;
  }

  // Reacts to receiving a message
  receive(message) {
    console.log(`${this.name} received: ${message}`);
  }
}

// Example usage:

const publisher = new Publisher();

// Creating subscribers
const subscriber1 = new Subscriber('Subscriber 1');
const subscriber2 = new Subscriber('Subscriber 2');

// Subscribing to the publisher
publisher.subscribe(subscriber1);
publisher.subscribe(subscriber2);

// Publisher sends a message
publisher.publish('Hello, world!');
```

Explanation:

- The **Publisher** has a list of **subscribers**. When it publishes a message, it sends it to each subscriber.
- The **Subscriber** has a receive method that logs the message it receives.

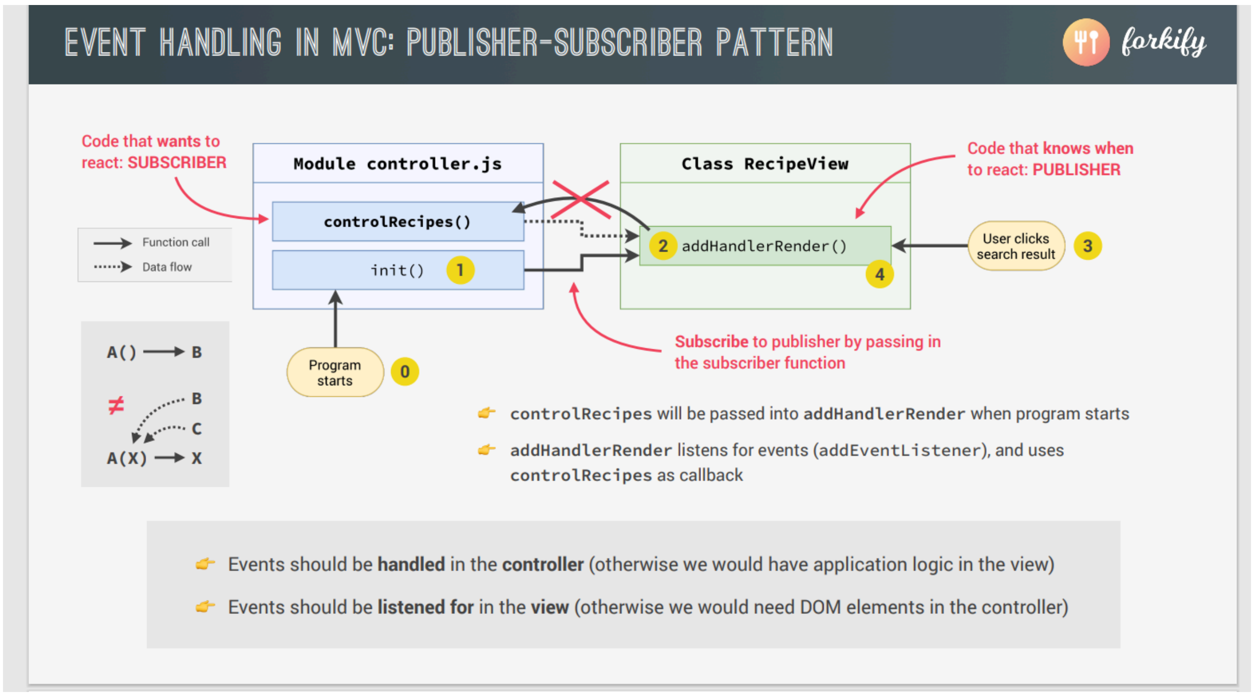
- **When the publisher publishes a message**, all subscribed entities (subscribers) receive it and react accordingly.

Output:

Subscriber 1 received: Hello, world!
Subscriber 2 received: Hello, world!

Why Use Publisher-Subscriber Pattern?

1. **Decoupling**: The publisher and subscriber don't need to know each other. This makes it easier to add, remove, or modify components without affecting others.
2. **Scalability**: Multiple subscribers can listen to the same event without the publisher knowing how many subscribers there are.
3. **Flexibility**: Subscribers can react to events asynchronously and in their own time.


EVENT HANDLING IN MVC: PUBLISHER-SUBSCRIBER PATTERN
forkify
Code that wants to react: SUBSCRIBER
Module controller.js
controlRecipes()
init() 1
Class RecipeView
2 addHandlerRender() 4
Code that knows when to react: PUBLISHER
User clicks search result 3
Function call
Data flow
Program starts 0
A() → B
≠ B C
A(X) → X
Subscribe to publisher by passing in the subscriber function

controlRecipes will be passed into addHandlerRender when program starts

addHandlerRender listens for events (addEventListener), and uses controlRecipes as callback

Events should be **handled** in the **controller** (otherwise we would have application logic in the view)

Events should be **listened for** in the **view** (otherwise we would need DOM elements in the controller)

What's Happening?

init Function:

```
const init = function () {
  recipeView.addHandlerRender(controlRecipes);
};
init();
```

- The init function initializes the application by registering a handler (controlRecipes) to respond to specific events (hashchange and load).
- This is done using recipeView.addHandlerRender().

addHandlerRender Method:

```
addHandlerRender(handler) {
  ['hashchange', 'load'].forEach(ev => window.addEventListener(ev, handler));
}
```

- This method sets up the **event listeners** for the hashchange and load events on the window object.
- It takes a handler function (here, controlRecipes) and attaches it to these events.

Publisher-Subscriber Relationship

Publisher:

The **publisher** is the window object in this case, which "publishes" events when certain actions occur:

- **hashchange**: Fired when the part of the URL after the # changes (e.g., navigating to index.html#recipe123).
- **load**: Fired when the page is fully loaded.

Subscriber:

The **subscriber** is the controlRecipes function, which is executed in response to the events published by the window object.

How the Pattern Works Here

1. **Subscription**:

    - Through the addHandlerRender method, the controlRecipes function subscribes to the hashchange and load events.
    - This subscription is established via window.addEventListener.
2. **Publishing**:
    - When the user changes the URL fragment (#hash) or reloads the page, the window object publishes these events (hashchange or load).
3. **Notification**:
    - The window object triggers the event listeners for the respective event. This causes the controlRecipes function to run and handle the new state or data.

Role of the Pattern

The **pub-sub pattern** decouples the logic for detecting changes (hashchange and load) from the logic for handling these changes (controlRecipes).

- The **Publisher** (window) doesn't know anything about the **Subscriber** (controlRecipes). It simply broadcasts events.
- The **Subscriber** doesn't need to know the internal workings of the publisher. It just subscribes to the events it cares about.

Advantages of This Approach

- **Decoupling**: The window object and controlRecipes function are independent of each other.
- **Scalability**: If more handlers need to respond to hashchange or load, they can be easily added without modifying the publisher.
- **Reusability**: The addHandlerRender method can be reused for different handlers or event types.

Implementing Error and Success Messages

The FormData method in JavaScript is used to create a new FormData object, which is designed to easily construct key-value pairs for use in HTTP requests, especially for sending form data using fetch or XMLHttpRequest.

How to Use FormData

Creating an Empty FormData Object

```
const formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

console.log(...formData.entries()); // Logs the key-value pairs
```

Creating a FormData Object from an HTML Form

```
const formElement = document.querySelector('form');
const formData = new FormData(formElement);

console.log(...formData.entries()); // Logs the form's key-value pairs
```

Key Methods of FormData

append(name, value)

Adds a new key-value pair to the FormData.

```
formData.append('username', 'JohnDoe');
```

set(name, value)

Sets a value for a key, overwriting if the key already exists.

```
formData.set('username', 'JaneDoe');
```

delete(name)

Deletes a key-value pair by name.

```
formData.delete('username');
```

get(name)

Retrieves the first value associated with a given key.

```
console.log(formData.get('username')); // 'JaneDoe'
```

getAll(name)
Retrieves all values associated with a given key (useful for multiple input fields with the same name).

```
console.log(formData.getAll('hobby')); // ['reading', 'coding']
```

has(name)
Checks if a specific key exists in the FormData.

```
console.log(formData.has('username')); // true
```

entries()
Returns an iterator with all key-value pairs.

```
for (const [key, value] of formData.entries()) {
  console.log(key, value);
}
```

keys()
Returns an iterator with all keys.

```
for (const key of formData.keys()) {
  console.log(key);
}
```

values()
Returns an iterator with all values.

```
for (const value of formData.values()) {
  console.log(value);
}
```

Standard Way to writing documentation

Visit : https://jsdoc.app/

```
/** */
```

1. Use a Documentation Tool

Tools like JSDoc are widely used in the JavaScript community. They parse comments in your code and generate structured HTML documentation.

2. Write Clear Inline Comments

Use comments to explain what the code does. Follow these standards:

- **Single-line comments**: Use // for brief explanations.
- **Multi-line comments**: Use /* ... */ for more detailed descriptions.

---

3. Follow a Commenting Format

Adopt a format like JSDoc for consistency. Here's a typical structure for documenting functions, classes, and modules:

Example: Documenting a Function

```
/**
 * Calculates the sum of two numbers.
 *
 * @param {number} a - The first number.
 * @param {number} b - The second number.
 * @returns {number} - The sum of the two numbers.
 *
 * @example
 * // Example usage:
 * const result = add(2, 3);
 * console.log(result); // 5
 */
function add(a, b) {
  return a + b;
}
```

Documenting Classes

```
/**
 * Represents a user.
 */
class User {
    /**
     * Creates a new user.
     *
     * @param {string} name - The user's name.
     * @param {string} email - The user's email address.
     */
    constructor(name, email) {
        this.name = name;
        this.email = email;
    }

    /**
     * Gets the user's details.
     *
     * @returns {string} - A string representation of the user's details.
     */
    getDetails() {
        return `Name: ${this.name}, Email: ${this.email}`;
    }
}
```

---

4. Include Examples

- Examples show how to use your code and clarify usage in real-world scenarios.
- Include them in your comments using the `@example` tag or a dedicated section in the documentation.

---

5. Document Modules and Files

Provide an overview at the beginning of each module or file:

```
/**
 * @module MathOperations
 * A module for basic mathematical operations.
 */
```

---

6. Write a README

Your README.md should:

- Describe the project and its purpose.
- Provide setup and installation instructions.
- Include usage examples and configuration options.

---

7. Use Version Control

Document the version of the API or library:

```
/**
 * @version 1.0.0
 */
```

---

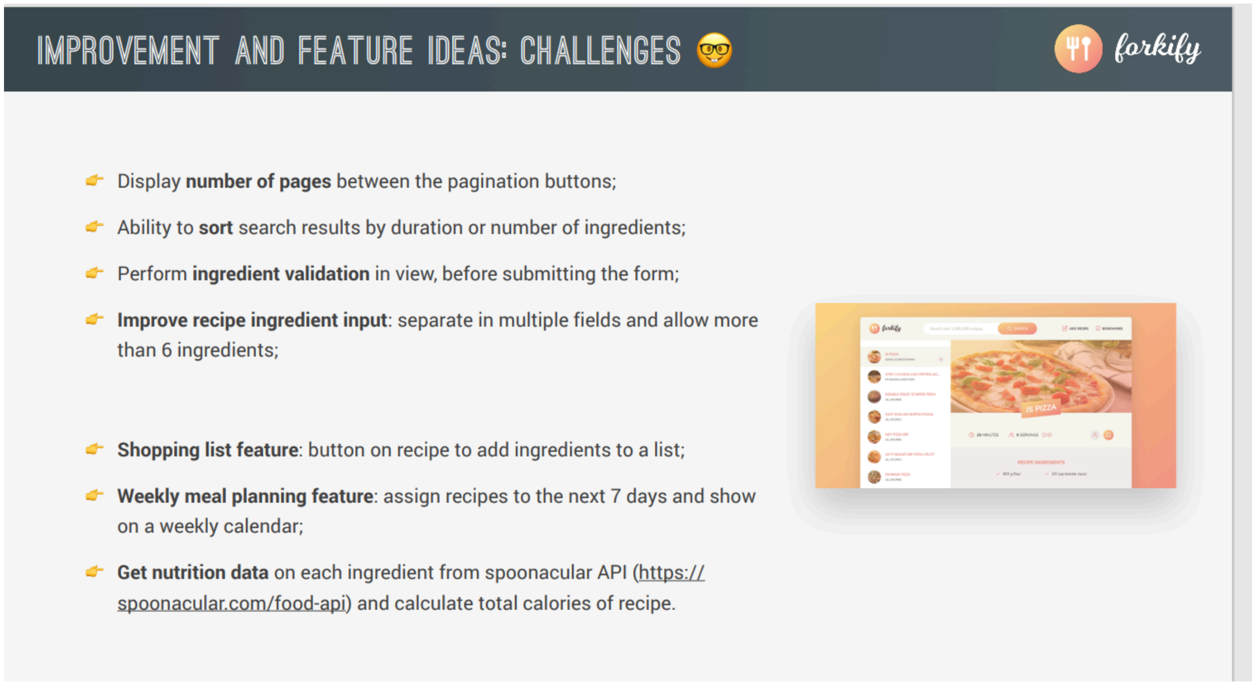
8. Make Documentation Accessible

- Host it using tools like GitHub Pages, Netlify, or documentation-specific tools like Docusaurus.
- Keep it up to date with changes in the codebase.

---

9. Review and Revise

- Ensure the documentation is easy to read and error-free.
- Collaborate with team members to ensure completeness.



IMPROVEMENT AND FEATURE IDEAS: CHALLENGES 🤓

- Display **number of pages** between the pagination buttons;
- Ability to **sort** search results by duration or number of ingredients;
- Perform **ingredient validation** in view, before submitting the form;
- **Improve recipe ingredient input**: separate in multiple fields and allow more than 6 ingredients;
- **Shopping list feature**: button on recipe to add ingredients to a list;
- **Weekly meal planning feature**: assign recipes to the next 7 days and show on a weekly calendar;
- **Get nutrition data** on each ingredient from spoonacular API (https://spoonacular.com/food-api) and calculate total calories of recipe.

Simple Deployment with Netlify

Netlify is a cloud-based company that focuses on providing tools and services to simplify the process of developing, deploying, and managing web applications and websites. Here's a breakdown:

Netlify is a modern cloud-based platform for hosting and deploying websites and web applications. It simplifies the process of building, deploying, and managing websites by integrating services like Continuous Deployment, serverless functions, and edge network delivery into a single platform. Here's an overview of what Netlify offers

1. **Remote-first**: Netlify operates as a distributed company, with employees working remotely from various locations rather than being centralized in a physical office.
2. **Development Platform**: Netlify provides an integrated environment that supports developers in building and deploying their web projects.
3. **Build**: It offers tools to automatically compile and prepare your code (e.g., HTML, CSS, JavaScript) for deployment.
4. **Deploy**: It enables easy hosting and launching of websites or web applications to make them live on the internet.
5. **Serverless Backend Services**: Netlify supports backend operations without needing to manage traditional servers. It allows developers to integrate serverless functions (like APIs) and database-like features directly into their projects.

Overall, Netlify is a service that streamlines the process of creating and running modern websites and apps, focusing on simplicity and speed.

What is Netlify?

- **Static Web Hosting:** It is primarily used for hosting static websites (HTML, CSS, and JavaScript), which do not rely on server-side rendering by default.
- **Continuous Deployment:** Integrated with Git repositories like GitHub, GitLab, and Bitbucket for automatic deployment whenever you push changes.
- **Serverless Functions:** Supports backend functionality through serverless functions using AWS Lambda. You can add dynamic features (e.g., API integrations, form submissions, authentication) without managing servers.
- **CDN (Content Delivery Network):** Automatically serves your site from a global CDN for faster load times.
- **Build Automation:** Can handle build processes for frontend frameworks like React, Vue, Angular, and Svelte directly on their platform.

Manual way of deploying Netlify

Here's the content rewritten as proper sentences:

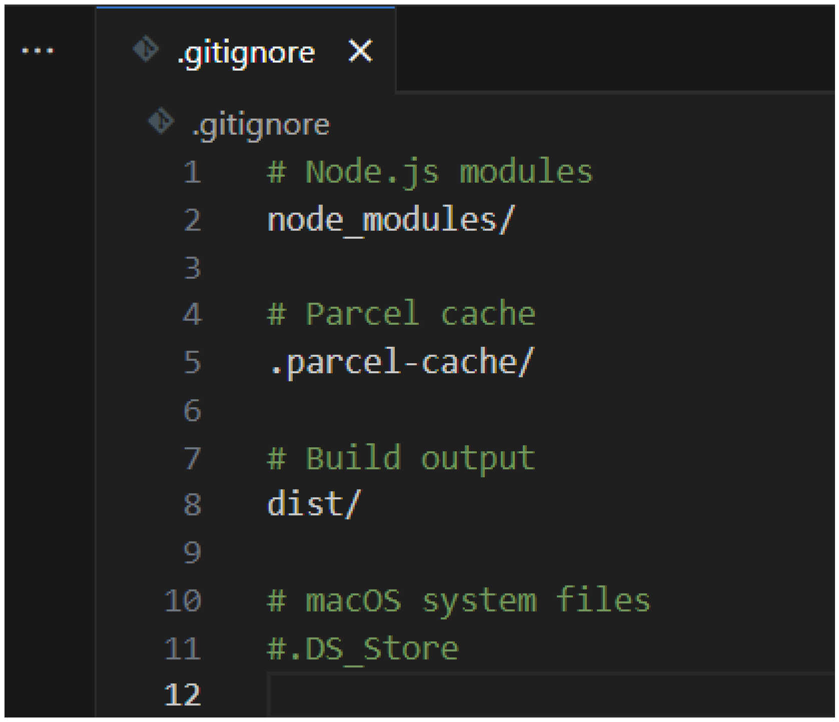
1. Remove or delete the .parcel-cache and dist folders from the current working directory in VS Code.
2. Rebuild the entire code using the command npm run build. Ensure that the "build": "parcel build index.html" script is already defined under the scripts section in the package.json file.
3. Open Netlify and create a free account if you don't already have one.
4. After the build process completes and the dist folder is created:
   - Open Netlify.
   - Deploy your site by dragging the dist folder into Netlify.
5. To change the site name (optional):
   - Go to **Site details** and select **Change site name**.
   - Enter a new name, such as forkify-ram.

Automatic way of deploying Netlify

1) Install Git (version control) from the official Git website.
2) Open the Command Prompt, navigate to the forkify-folder in VS Code, and run the command git init.

3) Visit the [official GitHub website]() and create an account if you don't already have one.
4) Configure your Git username and email:
    - Run the command `git config --global user.name "ram"` and press Enter.
    - Run the command `git config --global user.email "ram@gmail.com"` and press Enter.
5) Create a `.gitignore` file in the current folder. Inside the file, list the folders you want to ignore in the repository, such as:

```
node_modules
dist
.parcel-cache
```



```
# Node.js modules
node_modules/

# Parcel cache
.parcel-cache/

# Build output
dist/

# macOS system files
#.DS_Store

```

6) Run the following commands sequentially:
    - `git status` to check the current status of the repository.
    - `git add -A` to stage all changes for commit.
    - `git status` again to verify the staged changes.
    - `git commit -m "message"` to commit the changes with a descriptive message.
    - `git status` to confirm the working directory is clean after the commit.

    View the commit history by running `git log`.
7) To reset the repository to a specific commit, use the command:

```
git reset --hard <commit-id>
```

8) Create a new branch named new-feature by running:

```
git branch new-feature
```

9) Switch to the new-feature branch using:

```
git checkout new-feature
```

10) Merge the new-feature branch into the current branch by running:

```
git merge new-feature
```

Pushing to github

11) Create a new repository on GitHub and choose whether to make it public or private based on your preference.

    Link your local repository to the GitHub repository:

```
git remote add origin <repository-URL>
```

12) Verify the current branches in your repository by running:

```
git branch
```

13) Push the master branch to the GitHub repository:

```
git push origin master
```

14) Push the new-feature branch to the GitHub repository:

```
git push origin new-feature
```

Continuous integration with Netlify:

Link Your GitHub Repository to Netlify

1. **Log in to your Netlify account** (or sign up if you don't have one).
2. On the Netlify dashboard, click on **"New site from Git"**.
3. Select **"Link site to Git"**.
4. Click on **GitHub** to link your GitHub account.
5. Authenticate and allow Netlify to access your GitHub account.
6. Once authenticated, select the **repository** you want to deploy.

Configure the Build Settings

After selecting your repository, Netlify will ask for the following build settings:

Branch to deploy: Choose the branch you want to deploy (typically main or master).

Build command: This is the command that Netlify will use to build your project. For example, if you're using Parcel, you may enter:

```
npm run build
```

Publish directory: This is the folder where your built project files are located. For example, if using Parcel, this could be the dist folder. Enter:

```
dist
```

Set Up Continuous Deployment

Once the repository and build settings are configured, click **"Deploy site"**.

Netlify will now automatically build and deploy your site whenever you push changes to the connected GitHub repository. The continuous deployment process works as follows:

- Any code changes pushed to the selected branch (e.g., main) will trigger a new build on Netlify.
- The site will be redeployed automatically once the build is successful.