**32. Connecting Front-End to Back-End**

# **Introduction: Connecting Front-End to Back-End:** How Does the Frontend Communicate with the Backend?

## Definitions

**Server** - the abstract “thing” where the browser’s requests arrive. We don’t care about that detail to much here. It’s enough to know that the backend code is running on the server as well as other services. Request for the backend arrive at the server and are eventually passed on to your backend code.

**Backend** - the part of your web app which is not directly visible to the user. It receives requests and prepares data which is transmitted back to the user’s browser. Backend code is built to be running on a server and it’s never running on the user’s machine.

**Frontend** - the parts of your web application which are intended to be used directly by the the user’s browser. Code which is executed inside the browser, or markup which is interpreted while rendering a page. HTML, CSS and in-browser JavaScript are good examples for what I would consider to be part of the frontend concept. But only in their finished form. While the backend code can be assembling a HTML response, the final HTML arriving in the browser is meant here.

**Browser** - an application running on the user’s device. It sends out HTTP requests, receives responses, processes the received data, and uses it to render a viewable page. All of communication from the user’s side goes through their browser

## When does the browser talk to the backend?

HTTP requests arrive from the browser at the backend. Those requests may contain data in the HTTP headers or request body. The intent may be to request new data or to transmit user-created data to the backend.

HTTP requests, are constructed inside the user’s browser and sent off. There’s a response for each request, carrying information in the HTTP headers and the request body. Those responses arrive back at the backend arrive back at the user’s browser.

What needs to happen, to make the browser have to send out a request? Those requests which are fired because a user clicked a link or by JS code running in the background. But there are more possible triggers:

* The user enters a URL, which makes the browser go and request it.
* The browser reads the incoming HTML, and notices that there’s a resource it needs to load, such as a JS file, an image or a CSS file. It goes ahead and requests each with a single new HTTP request. Usually this happens while loading a website. (Those requests don’t have to go to the same backend, you could load JS from another site. People like to use CDNs for that as it’s pretty fast and convenient.)
* A user clicks on a plain-ol’ link the webpage is loaded and rendered. The browser knows that they need to navigate to a new page and requests the corresponding URL.
* JavaScript is executed on the site, and wants to have some data loaded in the background. Requests are being made, but the browser does that in the background. It doesn’t reload the whole page. That’s AJAX. Javascript can be triggered by a user clicking something, and it can tell the browser that the click wasn’t meant to navigate to a new page. This can be confusing.

## What do they send to each other?

The magic is within the HTTP request/response payloads. The backend usually responds with certain contents of the HTTP body:

* HTML-formatted responses
* other static files (CSS, JS, images, …)
* JSON-formatted data
* No body at all. Just a status code and header fields.

The frontend sends:

* Simple HTTP requests without a body
* Form data
* JSON-formatted data

## Let’s look at some examples

### Scenario: Just static content

If a browser requests something like http://example.com/style.css, this is usually handled by the web server (like Nginx) of the backend. The request is a GET for that resource, and the backend responds with a response, containing the content of the file.

Such static files can be served by the web application (application server) itself, but usually this is considered to be in poor taste. Web servers are really quick with this task, and web apps are kinda clumsy about it.

### Vanilla Django, using Django templates

The next case is about requesting dynamic content. From the browser side, it looks the same as requesting static content - there’s a request which is fired off to a URL like http://example.com/cute-puppies. The browser does not care which part of the backend will handle the request, it just wants answers. (NOW.)

The requests arrives at the server, is passed to the web server (Nginx for example), and the web server hands the request over to the app server (Django handled by Gunicorn). The web server simply does what it’s configured to do. Nothing magical here.

Anyway. Django gets the request, processes it, maybe looks up some data in a database and sends out a response.

*In more detail: Your Django app reacts to the request. The requested URL is interpreted using your urls.py configurations. The correct view is selected to handle the request. The view code might use a model to get data from the database, and renders out a template, passing data in a context object to it. Phew. This is what a “classical” Django project is. You get that out of the box, and this is a very good way to start for almost every web project out there.*

The resulting HTML is packaged in a response. The response contains an HTML page inside of the HTTP body.

The browser receives the response, and renders the DOM from the HTML. From there, the browser probably makes more requests to load those puppy images and some CSS so it can style the page. Those resources are static, and should be handled as in the first example case.

### Sprinkled in interactivity with jQuery, Vue.js or vanilla JS

Now, it gets a bit more fancy: JavaScript enters the picture in the frontend.

Imagine, that a dynamic page is loaded from the backend, just as described in the previous example. The returned HTML contains JS code inside a script tag.

The browser will render the page and start executing the JS code.

In this case, the code is only setting up some interactivity. Maybe to hide or show elements dynamically when a button is pressed, or to do on-demand communication with the backend.

The JS code, no matter which flavor, can ask for more requests to be sent to the backend. Fresh data can be requested this way, and the JS code takes care of handling it once it arrives without doing complete page reloads. HTTP requests triggered by JS code usually are made in the background.

We have the same page as before, but can trigger browser events without reloading the page. This makes for a more continuous user experience, as they don’t see the screen flash and new data loads smoothly, without breaking the UX flow.

### Frontend framework integrated: Vue.js for example

Now we get into JS framework territory. This case is the same as above, but the template returned by the backend is not necessarily complete. Chunks of the final page are missing from the initial HTML response.

Instead, it contains JS code which is responsible for filling in the blanks. In the previous example, the HTML markup was complete, and JS was only adding some interactivity.

The effect is, that the final HTML which is shown to the user is not the one which leaves the server.

The browser takes care of executing JS code once the page arrives, which in turn builds the rest of the page. You can do this conveniently with Vue.js - read more [here](https://build.vsupalov.com/vue-js-in-django-template).

### Single Page Application (SPA) - data is fetched afterwards

With the single-page-app architecture, a static page is loaded with a bunch of JS in it. Your backend code isn’t being used for this initial request. There’s nothing dynamic about it after all - it’s just about loading very simple HTML and fetching JS in the beginning.

The web server provides those. Django is usually not involved in serving the static assets in that case - the web server or another service such as AWS S3 takes care of those.

Once the HTML and JS has arrived at the browser, the JS is executed and starts making requests to the server for **DATA**. This is what the backend code provides - usually in the form of JSON responses.

The JavaScript code inside the static page takes care of loading data from your backend, and the DOM is built dynamically based on that data, starting with a blank page. The DOM elements which are dynamically assembled by JS are shown to the user. Your backend code does not know about how that page looks, and does not care much. It only cares about responding to data requests with JSON responses.

### Server-side rendering

In the previous case, the browser initially received an empty page with some JS linked from it.

As it turns out, this is something which search engines don’t like. They think those initially-empty pages are boring and this causes them to think less of your site.

They would much rather see a nice-looking, complete site.

This is what server-side rendering is for. The initial HTML and JS loaded on the server, and pre-rendered to HTML, but using frontend technology!

Essentially, there’s a part of the backend which pretends to be a browser for a while! It makes requests to your backend code and builds a HTML site by executing the JS part of the frontend code.

Once it’s done, the browser gets an HTML response, which was produced by JS code. The Django server is used to provide JSON data for that rendering step.

“Why all the hassle?” you may ask. Well, this is what you have to do if you want to have an SPA architecture, but still be friends with search engines.

Mainly, SSR is used for SEO reasons. It’s used to fix problems which are caused by delivering empty (or just incomplete), yet-to-be-assembled-by-JS HTML responses in the first place. There can be performance and caching upsides, but those usually don’t matter as much.

## In Conclusion

That’s it! I hope this article was useful to you, and you have a better overview of how the backend and the frontend communicate after reading it.

There are a lot of special cases, and things can get tricky, but in the end it’s only HTML and JSON going between the server and the user’s browser.

If you’re not sure which way to structure the communication between your frontend and backend - just keep it as simple as possible.

## Introduction

In this guide, I'll walk you through client-side rendering (CSR) in React.

A client-side rendered element is HTML content created by JavaScript. A few years back, JavaScript was used to do some basic DOM manipulations. Therefore, libraries like jQuery came into existence. But now it has become so powerful that we can build a complete UI system.

Today we can write HTML and CSS in JavaScript and render the page on demand, all thanks to client-side rendering.

## Playing with React

React is one of the great JavaScript libraries used in the frontend. A React application consists only of one HTML page, which is index.html, and to create other HTML content, we use JSX. React moves all the content into one HTML file. This is known as Single Page Application, or SPA, where JavaScript handles the rendering logic. Every SPA framework uses its own routing to display HTML content on different pages or routes.

But if we talk about JSX, then a question arises.: "How does it work internally?"

JSX is first converted into plain JavaScript because browsers support only JavaScript. After that, JavaScript creates the HTML content and renders it on the user's screen.

I'll give you a brief idea of the internal structure of React in the following section, which will give you clarity on CSR.

## Creating HTML with JavaScript

Below is a sample code which will give you an idea of how to create an HTML element with JavaScript.

1function elt(type, props, ...children) {

2 let dom = document.createElement(type);

3 if (props) Object.assign(dom.props);

4

5 for (let child of children) {

6 if (typeof child != string) dom.appendChild(child);

7 else dom.appendChild(document.createTextNode(child));

8 }

9 return dom;

10}

js

In the above snippet, I have written a function with three arguments: type, props, and children, with rest operator that will send any number of children.

The createElement() method of the document creates an element on the DOM.

1document.createElement("input");

javaScript

The above piece of code will generate an input tag in HTML.

The appendChild() method of the document appends the created element to the block.

1let dom = document.crateElement("div");

2let dom1 = document.createElement("input");

3dom.appenChild(dom1);

javaScript

The above code will output the following HTML content :

1<div>

2 <input />

3</div>

html

Now you can see perfectly how to create one tag under another tag using appendChild() method.

The createTextNode() method of the document helps you generate a string label on the DOM.

1let dom = document.crateElement("div");

2let label = docuemnt.createTextNode("Name:");

3dom.appenChild(label);

4let dom1 = document.createElement("input");

5dom.appenChild(dom1);

javaScript

The above code generates the HTML, as follows :

1<div>Name:<input /></div>

html

Now you're ready to learn about the elt() method. You can generate the same HTML of the name label and <input /> under the <div> using elt() method.

1elt(

2 "div",

3 {},

4 "Name",

5 elt("input", {

6 type: text,

7 value: ""

8 })

9);

javaScript

Output:

1<div>Name:<input /></div>

html

This is how to create an HTML element using JavaScript. Let's now take a look at the pros and cons of client-side rendering.

## Pros of CSR

### High Performance

CSR generates on-demand HTML. It will not refresh or re-render the whole page, as with regular HTML pages. It is just pretending to be a separate page, but it renders the content on a single page. This saves a lot of both computation power and RAM, so it gets quicker results than server-side rendering (SSR).

### Speed

CSR generates the HTML required to be displayed. This means DOM only contains enough code which is expected to be displayed by the HTML content. So DOM can easily handle a chunk of elements with the hide and show events. Although DOM has to handle more code, it doesn't take much time to render. Due to lazy loading, CSR becomes much faster than server-side rendering.

### Reusable components

With CSR, we can reuse UI components across multiple pages or routes without having to request the server each time. This enhances usability and on-page performance.

But on the other hand, there are some disadvantages of CSR which we will discuss below.

## Cons of CSR

### Slow at First

CSR loads the whole JavaScript the first time, then calls for API to get the data from the database and generate the HTML as per data. But loading the data for the first time takes a bit more time than server-side rendering.

### SEO problem

s

SEO stands for search engine optimization. CSR requires a two-wave process for JS rendering and indexing in a browser, generally by Google.

The first wave requests the source code, crawls, and then indexes the presented HTML. But in CSR we don't have much of the HTML because it takes time to convert from JavaScript to HTML.

In the second wave, after all resources become available, the browser returns the additional support and index to the search engine. This is not a problem with server-side rendering because in server-side rendering, the HTML is available for the first time itself.

### Caching Issue

Since the HTML is not available in the initial render, browsers cannot cache the HTML structure of the page. One way to avoid this issue is to cache the JavaScript, but this may prove to be costly as JavaScript files can take up a lot of space in the browser's memory.

## Conclusion

CSR has multiple benefits over SSR, but there are some pitfalls we can't deny. But after the first load, it becomes very smooth and user-friendly. It gives high performance and high speed and doesn't take much RAM to run.

I hope this guide was very informative and a gateway to your client-side crusades.

14

# What’s Server Side Rendering and do I need it?

[Bartosz Szczeciński](https://medium.com/@baphemot?source=post_page-----cb42dc059b38--------------------------------)

[Bartosz Szczeciński](https://medium.com/@baphemot?source=post_page-----cb42dc059b38--------------------------------)

[Sep 1, 2018·7 min read](https://medium.com/@baphemot/whats-server-side-rendering-and-do-i-need-it-cb42dc059b38?source=post_page-----cb42dc059b38--------------------------------)

Introduction of modern JavaScript frameworks / libraries that focus on creating interactive websites or Single Page Applications the way that pages are displayed to a visitor has changed a lot.

Before the advent of applications fully generated by JS, in the browser, HTML was returned in response to the HTTP call — be it by returning a static HTML content file, or by processing the response via server side languages (such as PHP, Python or Java) and responding in a more dynamic way.

A solution like this allows us to deliver responsive sites that work a lot faster than standard request-response model by removing the “request travel time”.

A typical response sent by the server when requesting a React site will look something like this:

<!DOCTYPE html>  
<html lang="en">  
 <head>  
 <meta charset="utf-8">  
 <link rel="shortcut icon" href="/favicon.ico">  
 <title>React App</title>  
 </head>  
 <body>  
 <div id="root"></div>  
 <script src="/app.js"></script>  
 </body>  
</html>

After fetching this response our browser will also fetch the app.js “bundle” which contains our application and after a second or two render the complete page.

At this moment we can use our browser HTML Inspector to view the complete rendered HTML, but when we view the actual source we will see nothing more than the above HTML code.

# **Why is this an issue?**

While this behavior will not be a problem for most our users or when developing the application it will become an issue if:

* the end user is using a **slow internet connection**, especially mobile connection,
* the end user is running an **underpowered device**, for example an older generation mobile device,

If your target demographic falls into one of those groups they will have a bad experience using the site — for example will have to wait a lot longer, staring at the “Loading …” (or even worse — a blank screen) message.

“OK, but my target demographic is not in any of those groups, should I still care?”

There’s one more important thing you should consider when going with a client-side rendered app: **search engines** and **social networks presence**.

Currently, of all the search engines only Google has limited capabilities to render and JS site before indexing it. In addition, while Google will be able to render the index page of your website it is known to have issues navigating around sites with a router implementation.

This means that **your site will have very hard time trying to get top position in the search results** in anything but Google.

The same issue is visible in social platforms such as **Facebook**— when sharing a link to your site neither the title nor the thumbnail will render properly.

# **How to solve the issue**

There are a few ways we can solve the issue.

## **A — Consider having your key pages as static**

When you’re creating an platform that requires the users to login, and not providing the content to not-signed in users you might decide to**create your public facing sites (like the index, “about us”, “contact us” etc.) pages as static HTML**, and not have them rendered by JS.

Since your content is gated by login requirements it will not be indexed by search engines or shared in social media.

## **B — Generate parts of your application as HTML pages when running the build process**

Libraries like [react-snapshot](https://github.com/geelen/react-snapshot) can be added to your project, used to generate HTML copies of your application pages and save them to a specified folder. You can then deploy this folder alongside your JS bundle. This way, the HTML will be served along with the response allowing your site to be accessible by users with JavaScript disabled, search engines etc.

In most cases, configuration of react-snapshot is as simple as adding the library to your project, and altering the build script by calling it at the end:

"build": "webpack && react-snapshot --build-dir static"

The downside of this solution is that all the content we want to generate must be available at build time — we can’t query any APIs to get it, we can’t pre-generate content that depends on user provided data (e.g. as URL).

## **C — Create a server-side rendered application in JS**

One of the big selling point of the current generation of JS applications is the fact, that they can be ran on both the client (browser) and on server — this allows us to generate HTML for pages that are more dynamic — which content is not known at build time. This is often referred to as “isomorphic” or “universal” application.

Two of the most popular solutions that provide SSR for React are:

* next.js — <https://github.com/zeit/next.js/>
* Gatsby — <https://github.com/gatsbyjs/gatsby>

# **Create your own custom SSR implementation**

Important: If you’re willing to try to and create your own SSR implementation for your React applications, you’re going to need to be able to run a node backend for your server. You will not be able to deploy this solution to static host like github pages.

First step we’re going to take is to create an application just like you would with any other React application.

Let’s create our entry point:

// index.js  
import React from 'react';  
import { render } from 'react-dom';  
import App from './App.js';render(<App />, document.getElementById('root'));

And the App component:

// App.js  
import React from 'react';const App = () => {  
 return (  
 <div>  
 Welcome to SSR powered React application!  
 </div>  
 );  
}

And our “shell” to load our application:

// index.html  
<!doctype html>  
<html>  
 <head>  
 <meta charset="utf-8" />  
 </head>  
 <body>  
 <div id="root"></div>  
 <script src="/bundle.js"></script>  
 </body>  
</html>

As you see our application is quite simple. We’re not going to go through all the steps needed to generate the proper webpack+babel setup in this article.

If you run the application in its current state, you will notice the welcome message on screen, and viewing the source you will see the content of index.html file, without the actual welcome message. To solve this issue let’s add server side rendering. First, we will need to add 3packages:

yarn add express pug babel-node --save-dev

Express is a powerful webserver for node, pug is a templating engine we can use with express, and babel-node is a wrapper for node, which allows it to perform code transpilation on the fly.

First, we will make a copy of our index.html file, and save it as index.pug:

// index.pug  
<!doctype html>  
<html>  
 <head>  
 <meta charset="utf-8" />  
 </head>  
 <body>  
 <div id="root">!{app}</div>  
 <script src="bundle.js"></script>  
 </body>  
</html>

You’ll notice the file is basically the same, save for !{app} being inserted into the HTML. This is a pug variable, which will be later on replaced by the actual HTML.

Let’s now create our server:

// server.jsimport React from 'react';  
import { renderToString } from 'react-dom/server';  
import express from 'express';  
import path from 'path';import App from './src/App';const app = express();  
app.set('view engine', 'pug');  
app.use('/', express.static(path.join(\_\_dirname, 'dist')));app.get('\*', (req, res) => {  
 const html = renderToString(  
 <App />  
 ); res.render(path.join(\_\_dirname, 'src/index.pug'), {  
 app: html  
 });  
});app.listen(3000, () => console.log('listening on port 3000'));

Let’s step through the file.

import { renderToString } from 'react-dom/server';

react-dom library contains a separate, named export renderToString which works similarly to the render we know, but instead of rendering to the DOM, it renders the HTML as string.

const app = express();  
app.set('view engine', 'pug');  
app.use('/', express.static(path.join(\_\_dirname, 'dist')));

We’re creating a new express server instance, and letting it know we want to use the pug template engine. We could get away with reading the file and doing a search & replace, but that’s really ineffective and can cause issues with multiple access to the file system or caching issues.

In the last line, we’re instructing express to look for file in the dist folder, and if a request (e.g. /bundle.js) matches a file, that is present in that folder, reply with it.

app.get('\*', (req, res) => {  
});

Now in turn we ask express to add a handler for any unmatched URL — this includes our non-existing index.html file (remember, we renamed it to index.pug and also it’s not available in the dist folder).

const html = renderToString(  
 <App />  
);

Using renderToString we render our application. This code looks exactly like our entry point, but it does not have to.

res.render(path.join(\_\_dirname, 'src/index.pug'), {  
 app: html  
});

Once we have the rendered HTML, we tell express to respond by rendering the index.pug file, and replacing the app variable, with the HTML we received.

app.listen(3000, () => console.log('listening on port 3000'));

Last part is actually starting the server, and making it listen on port 3000.

All that’s left to do is to add a proper script topackage.json:

"scripts": {  
 "server": "babel-node server.js"  
}

Now, once we call yarn run server we should receive a confirmation, that the server is indeed running. Navigate your browser to [http://localhost:300](http://localhost:3000/)0 where you should again see your application. But this time if you view the source you should see:

<!doctype html>  
<html>  
 <head>  
 <meta charset="utf-8" />  
 </head>  
 <body>  
 <div id="root">div data-reactroot="">Welcome to SSR powered React application!</div></div>  
 <script src="bundle.js"></script>  
 </body>  
</html>

If that is the case, it means that server-side rendering is working as expected, and you can start to grow your application!

## **Why do we still need bundle.js?**

In case of our super-simple application, inclusion of bundle.js is not needed — without it, our application would still work. But in case of real-life applications, you’d still want to include the file.

This allows the browsers that know how to handle JavaScript to take over and perform the future interaction with your page on the client side, and those that do not know how to parse JS to navigate the page with the actual HTML returned by the server.

## **Things to keep in mind**

While SSR might look simple enough, when developing the application you need to watch out for some topics, which might not be initially clear:

* any state generated on the server side will not be passed to the client application state; that means that if your backend fetches some data and uses it to render HTML, it will not be placed in the this.state that the browser sees
* componentDidMount is not called on the server — this means that none of your data-fetching that you’re used to place there will not be called; this is generally a good thing, as you should be providing the data as props if you need it. Remember that you need to delay your render (calling res.render) until the data is fetched, which might introduce some delay for your visitors
* if you’re using a react router (e.g. [@reach/router](https://reach.tech/router/server-rendering) or [react-router](https://reacttraining.com/react-router/web/guides/server-rendering)) you need to make sure that the proper URL is passed to the application when it’s rendered on the server — be sure to check the proper documentation!

Which of the following is NOT an advantage of server-side rendering?

Compared to single page applications, the HTML sent to the browser will be larger for server-side rendered apps because it is the fully assembled page, rather than skeleton HTML where content is fetched afterwards on the client-side. Smaller HTML response to send to the browser.

Imagine you have a template file called **index.pug** that contains the variable **content**. Complete the code for the React application below so it would render the **Welcome** component as the content on your page. Assume all the needed imports and setup are taken care of.

app.get('\*', (req, res) => {  
  const info = renderToString(  
    <Welcome />  
  );  
   
  res.render(path.join(\_\_dirname, 'src/index.pug'), {  
    content: info  
  });  
});

## What is SSR and why should you care?

SSR stands for Server Side Rendering. I will talk mostly about React, but I guess it will make sense for other frameworks too.

You need SSR if you care about:

* **First meaningful paint**. SSR alone doesn't guarantee good results. You also need critical CSS and proximity to the client etc.
* **SEO** and support other bots like Twitter and Facebook
* **Graceful degradation**. For this one, you need to make sure your service is usable without JS

## What is hard about it?

SSR is like a new dimension. Whatever you use you will need to reconfigure it for SSR.

* Do you use componentDidMount to fetch the data? You need to use something like getInitialProps (from next.js or after.js) or state management library like Redux to make it work on the server
* Do you use Router? You need to configure it on the server
* Do you use i18n? You need to configure it on the server
* Do you use HMR? You will need to reload code for browser and for the server
* Do you use react-helmet? You need to configure it on the server
* Do you use react-loadable? You need to configure the server to pass used modules, so the client can preload them
* Do you use Redux? You need to serialize store and pass it to the client
* Do you use CSS-in-JS? You need to configure it to generate critical CSS on the server and inline it in HTML response

## SEO

There are 3 options here:

* SSR
* Prerendering, like react-snap, react-static, gatsby etc.
* Prerendering on the demand, like rendertron, puppetron, pupperender etc.

# MVC: Model, View, Controller

**App organization explained**

## Background

MVC is short for Model, View, and Controller. MVC is a popular way of organizing your code. The big idea behind MVC is that each section of your code has a purpose, and those purposes are different. Some of your code holds the data of your app, some of your code makes your app look nice, and some of your code controls how your app functions.

MVC is a way to organize your code’s core functions into their own, neatly organized boxes. This makes thinking about your app, revisiting your app, and sharing your app with others much easier and cleaner.

## The parts of MVC

**Model:** Model code typically reflects real-world things. This code can hold raw data, or it will define the essential components of your app. For instance, if you were building a To-do app, the model code would define what a “task” is and what a “list” is – since those are the main components of a todo app. **View:** View code is made up of all the functions that directly interact with the user. This is the code that makes your app look nice, and otherwise defines how your user sees and interacts with it. **Controller:** Controller code acts as a liaison between the Model and the View, receiving user input and deciding what to do with it. It’s the brains of the application, and ties together the model and the view.

## An analogy

MVC is a way to think about how an web application works.

It’s kind of like how you make Thanksgiving dinner. You have a fridge full of food, which is like the Model. The fridge (Model) contains the raw materials we will use to make dinner.

You also probably have a recipe or two. A recipe (assuming you follow it exactly) is like the Controller of Thanksgiving dinner. Recipes dictate which stuff in the fridge you’ll take out, how you’ll put it together, and how long you need to cook it.

Then, you have table-settings, silverware, etc., which are what your hungry friends and family use to eat dinner. Table-top items are like the View. They let your guests interact with your Model and Controller’s creation.

## MVC in the real-world

MVC is helpful when planning your app, because it gives you an outline of how your ideas should be organized into actual code.

For instance, let’s imagine you’re creating a To-do list app. This app will let users create tasks and organize them into lists.

The **Model** in a todo app might define what a “task” is and that a “list” is a collection of tasks.

The **View** code will define what the todos and lists looks like, visually. The tasks could have large font, or be a certain color.

Finally, the **Controller** could define how a user adds a task or marks another as complete. The Controller connects the View’s add button to the Model, so that when you click “add task,” the Model adds a new task.

## Wrapping up

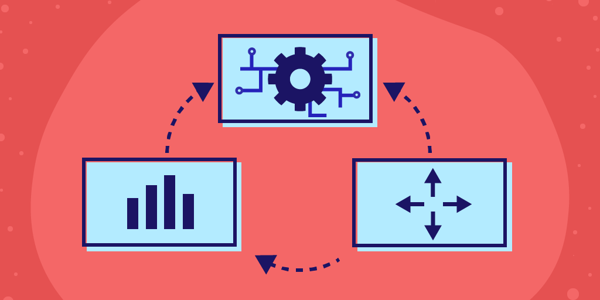
MVC is a framework for thinking about programming, and for organizing your program’s files. To signify the idea that your code should be organized by its function, developers will create folders for each part of MVC. (The idea that apps should be divided based on the function of each part of the code is sometimes referred to as separation of concerns.) If you’ve looked at Codecademy’s Ruby on Rails course, you might have noticed that there is a folder for each part of MVC within every Rails app it introduces.

MVC gives you a starting place to translate your ideas into code, and it also makes coming back to your code easier, since you will be able to identify which code does what. In addition, the organizational standard MVC promotes makes it easy for other developers to understand your code.

Thinking about how code interacts with other code is a significant part of programming, and learning to collaborate with other developers is an important skill. Taking the time to think about how your app fits into the MVC framework will level-up your skills as a developer by teaching you both. It’ll also make your apps better!

|[Blog Home](https://www.educative.io/blog)

# MVC Architecture in 5 minutes: a tutorial for beginners



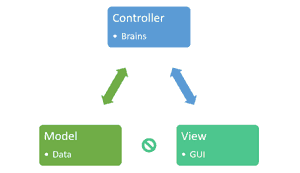
Model View Controller is a predictable [software design pattern](https://www.educative.io/blog/how-to-design-a-web-application-software-architecture-101) that can be used across many frameworks with many programming languages, commonly [Python](https://www.educative.io/blog/web-development-in-python), [Ruby](https://www.educative.io/blog/ruby-on-rails), [PHP](https://www.educative.io/blog/php-tutorial-from-scratch), JavaScript, and more. It is popularly used to design web applications and [mobile apps](https://www.educative.io/blog/how-to-develop-an-android-app).

This blog post defines the concept of a Model-View-Controller (MVC) software design pattern and does a basic example Model-View-Controller in [JavaScript/HTML/CSS](https://www.educative.io/blog/beginner-guide-to-web-dev).

## What is MVC Architecture?

The architecture components of the [MVC pattern](https://www.educative.io/edpresso/what-is-the-mvc-architecture) are designed to handle different aspects of an application in development. The MVC design pattern serves to separate the presentation layer from the business logic.Why do developers care about MVC? MVC is popular in app and web development, and it’s one of the most widely used software design patterns for app and web development. The Model View Controller design pattern separates concerns into one of 3 buckets:

* Model
* View
* Controller



Model View Controller Design Pattern: Separation of Concerns

The Model View Controller architectural pattern separates concerns into one of 3 buckets:

1. **Model:** stores & manages data.

Often a [database](https://www.educative.io/blog/database-design-tutorial), in our quick example we’ll use local web storage on a browser to illustrate the concept.

1. **View:** Graphical User Interface

The view is a visual representation of the data- like a chart, diagram, table, form.

The view contains all functionality that directly interacts with the user - like clicking a button, or an enter event.

1. **Controller:** Brains of the application.

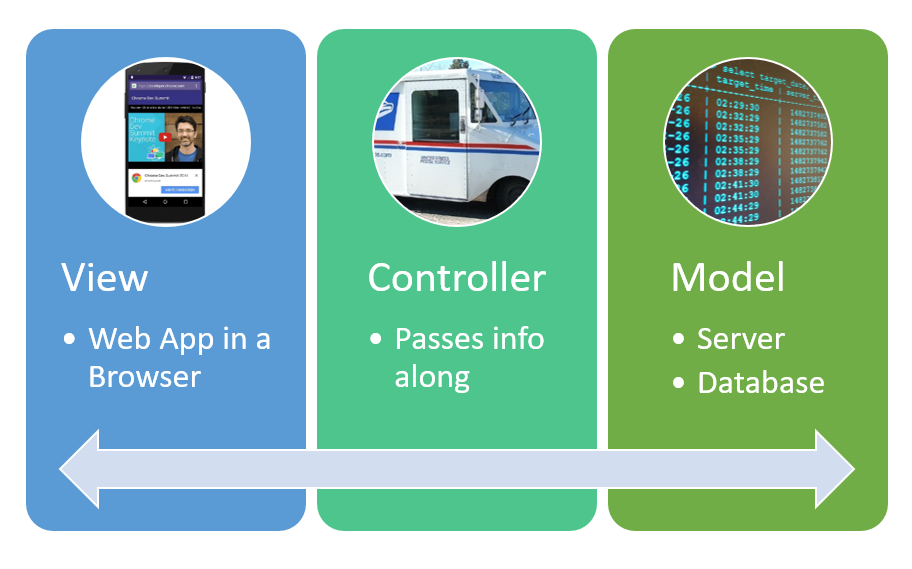
The controller connects the model and view. The controller converts inputs from the view to demands to retrieve/update data in the model.

The controller receives input from view, uses logic to translate the input to a demand for the model, the model grabs the data, the controller passes data from the model back to the view for the user to see in a nice display.

## Benefits of MVC

* Traditionally used for Graphical user interfaces (GUIs)
* Popular in web applications
* MVC responsibilities are divided between the client & server, compatible with web application architecture
* MVC is helpful design pattern when planning development
* Separation of Concerns: that code is divided based on function to either the model, view, or controller bucket
* Works well with Ruby on Rails
* Loosely Coupled
* Removes unnecessary dependencies
* Reuseable without modification
* MVC makes model classes reusable without modification
* Code reuse
* Extendable code
* High Cohesion
* Easier to maintain or modify
* Supports Multiple views
* Each part can be tested independently (Model, view, controller)

## Web App as a MVC



You can think of a web application as a Model View Controller design. MVC is popular in web applications, one of the reasons is because responsibilities are divided between the client & server.

## Separation of Concerns

MVC design allows for Separation of Concerns - dividing the logic up between the 3 buckets, so that each bucket can act independently.

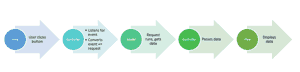
The model, view, and controller don’t depend on each other. Why does this matter? Generally, software is worked on by teams - a team might have a designer, engineer, and database architect. Separation of concerns means each team member can work on their piece at the same time, because logic has been separated into buckets. Separation of concerns is also great for maitenance - developers can fix a bug in one piece of code, without having to check out the other pieces of code.

### Loosely Coupled

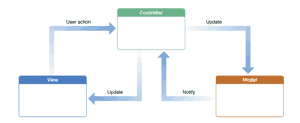
Loosely coupled means that each piece: the model, view and controller, act independently of eachother.

Developers can modify one of the pieces, and the other 2 pieces should keep working and not require modifications. When designing MVC software – the logic in each of the three buckets is independent. Everything in View acts independently of the model – and vice verse, the view won’t have any logic dependent on the model.

Making independent models and views makes code organization simple and easy to understand and keeps maintenance easier. Programmers can fix a bug in the view without changing the model code.



MVC Design: how a user sees MVC



MVC with User Action

The pictures above show what happens in a MVC web app when a user clicks a button, from the perspective of the user.

# Introduction

Express.js is great frameworks for making a node.js REST APIs however it doesn’t give you any clue on how to organizing your node.js project.

While it may sound silly, this is a real problem.

The correct organization of your node.js project structure will avoid duplication of code, will improve stability, and potentially, will help you scale your services if is done correctly.

This post is extense research, from my years of experience dealing with a poor structured node.js project, bad patterns, and countless hours of refactoring code and moving things around.

If you need help to align your node.js project architecture, just drop me a letter at sam@softwareontheroad.com

# Table of contents

* [The folder structure 🏢](https://softwareontheroad.com/ideal-nodejs-project-structure/#folder)
* [3 Layer architecture 🥪](https://softwareontheroad.com/ideal-nodejs-project-structure/#architecture)
* [Service Layer 💼](https://softwareontheroad.com/ideal-nodejs-project-structure/#service)
* [Pub/Sub Layer ️️️️🎙️️](https://softwareontheroad.com/ideal-nodejs-project-structure/#pubsub)
* [Dependency Injection 💉](https://softwareontheroad.com/ideal-nodejs-project-structure/#di)
* [Unit Testing 🕵🏻](https://softwareontheroad.com/ideal-nodejs-project-structure/#test)
* [Cron Jobs and recurring task ⚡](https://softwareontheroad.com/ideal-nodejs-project-structure/#cron)
* [Configurations and secrets 🤫](https://softwareontheroad.com/ideal-nodejs-project-structure/#configs)
* [Loaders 🏗️](https://softwareontheroad.com/ideal-nodejs-project-structure/#loaders)
* [Example repository](https://github.com/santiq/bulletproof-nodejs)

# The folder structure 🏢

Here is the node.js project structure that I’m talking about.

I use this in every node.js REST API service that I build, let’s see in details what every component do.

src

│ app.js # App entry point

└───api # Express route controllers for all the endpoints of the app

└───config # Environment variables and configuration related stuff

└───jobs # Jobs definitions for agenda.js

└───loaders # Split the startup process into modules

└───models # Database models

└───services # All the business logic is here

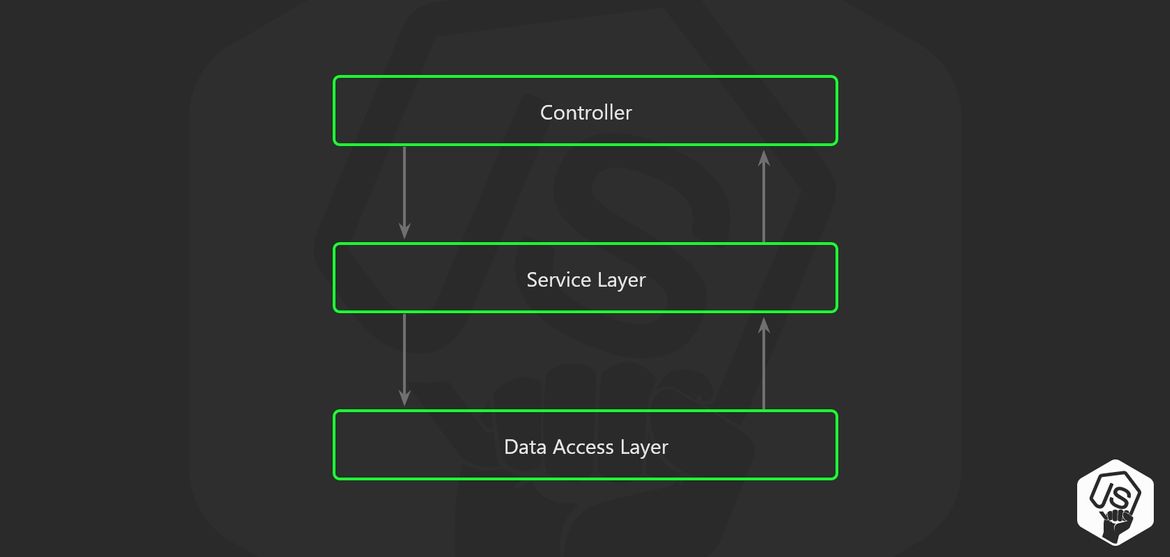
└───subscribers # Event handlers for async task

└───types # Type declaration files (d.ts) for Typescript

It is more than just a way of ordering javascript files…

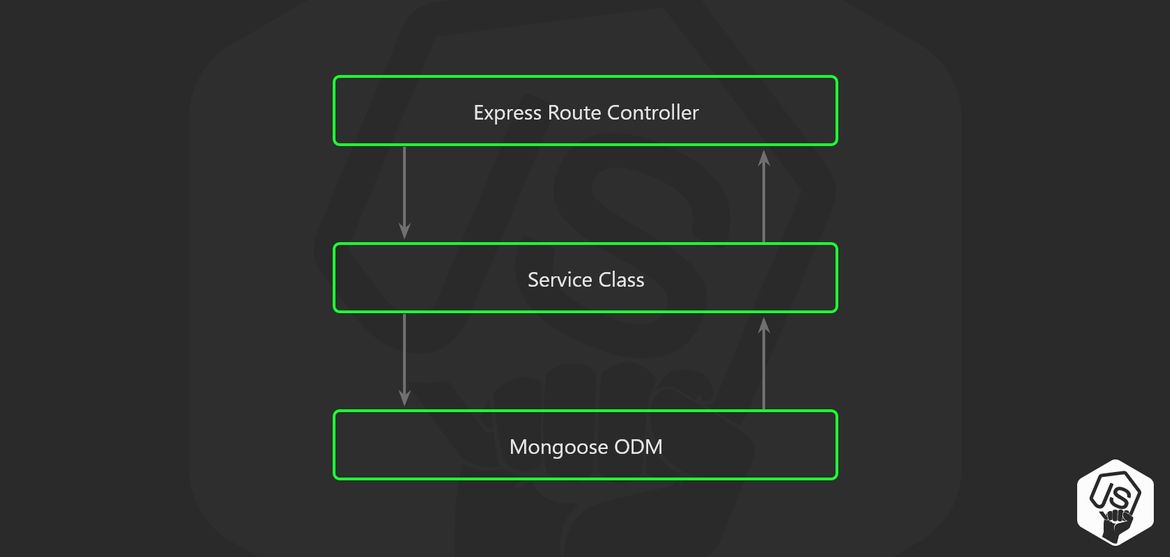
# 3 Layer architecture 🥪

The idea is to use the **principle of separation of concerns** to move the business logic away from the node.js API Routes.

[](https://softwareontheroad.com/static/122dab3154cb7e417bbb210bbce7ca01/8299d/server_layers.jpg)

Because someday, you will want to use your business logic on a CLI tool, or not going far, in a recurring task.

And make an API call from the node.js server to itself it’s not a good idea…

[](https://softwareontheroad.com/static/1a21f74cfc4c965f00324afd39642b9f/8299d/server_layers_2.jpg)

## ☠️ Don’t put your business logic inside the controllers!! ☠️

You may be tempted to just use the express.js controllers to store the business logic of your application, but this quickly becomes spaghetti code, as soon as you need to write unit tests, you will end up dealing with complex mocks for req or res express.js objects.

It’s complicated to distingue when a response should be sent, and when to continue processing in ‘background’, let’s say after the response is sent to the client.

Here is an example of what not to do.

route.post('/', async (req, res, next) => {

// This should be a middleware or should be handled by a library like Joi.

const userDTO = req.body;

const isUserValid = validators.user(userDTO)

if(!isUserValid) {

return res.status(400).end();

}

// Lot of business logic here...

const userRecord = await UserModel.create(userDTO);

delete userRecord.password;

delete userRecord.salt;

const companyRecord = await CompanyModel.create(userRecord);

const companyDashboard = await CompanyDashboard.create(userRecord, companyRecord);

...whatever...

// And here is the 'optimization' that mess up everything.

// The response is sent to client...

res.json({ user: userRecord, company: companyRecord });

// But code execution continues :(

const salaryRecord = await SalaryModel.create(userRecord, companyRecord);

eventTracker.track('user\_signup',userRecord,companyRecord,salaryRecord);

intercom.createUser(userRecord);

gaAnalytics.event('user\_signup',userRecord);

await EmailService.startSignupSequence(userRecord)

});

React information:

## **SINGLE REACT FILE**

The first step follows the rule: One file to rule them all. Most React projects start with a src/ folder and one src/App.js file with an App component. At least that's what you get when you are using [create-react-app](https://www.robinwieruch.de/react-js-macos-setup). This App [function component](https://www.robinwieruch.de/react-function-component/) just renders something:

import React from 'react';

const App = () => {

const title = 'React';

return (

<div>

<h1>Hello {title}</h1>

</div>

);

}

export default App;

Eventually this component adds more features, it naturally grows in size, and needs to extract parts of it as standalone React components. Here we are extracting a [React list component](https://www.robinwieruch.de/react-list-component) with another child component from the App component:

import React from 'react';

const list = [

{

id: 'a',

firstname: 'Robin',

lastname: 'Wieruch',

year: 1988,

},

{

id: 'b',

firstname: 'Dave',

lastname: 'Davidds',

year: 1990,

},

];

const App = () => <List *list*={list} />;

const List = ({ list }) => (

<ul>

{list.map(item => (

<ListItem *key*={item.id} *item*={item} />

))}

</ul>

);

const ListItem = ({ item }) => (

<li>

<div>{item.id}</div>

<div>{item.firstname}</div>

<div>{item.lastname}</div>

<div>{item.year}</div>

</li>

);

Whenever you start with a new React project, I tell people it's fine to have multiple components in one file. It's even fine in a larger React application, whenever one component is strictly tight to another one. However, in this scenario, eventually your one file will not be sufficient anymore for your React project. That's when we transition to step two.

## **MULTIPLE REACT FILES**

The second step follows the rule: Multiple files to rule them all. Take for instance our previous App component with its List and ListItem components: Rather than having everything in one src/App.js file, we can split these components up into multiple files. You decide how far you want to take it here. For example, I would go with the following folder structure:

- src/

--- App.js

--- List.js

While the src/List.js file would have the implementation details of the List and ListItem components, it would only [export](https://www.robinwieruch.de/javascript-import-export) the List component from the file as public API to this file:

const List = ({ list }) => (

<ul>

{list.map(item => (

<ListItem *key*={item.id} *item*={item} />

))}

</ul>

);

const ListItem = ({ item }) => (

<li>

<div>{item.id}</div>

<div>{item.firstname}</div>

<div>{item.lastname}</div>

<div>{item.year}</div>

</li>

);

export default List;

Then the src/App.js file can [import](https://www.robinwieruch.de/javascript-import-export) the List component and use it:

import React from 'react';

import List from './List';

const list = [

{

id: 'a',

firstname: 'Robin',

lastname: 'Wieruch',

year: 1988,

},

{

id: 'b',

firstname: 'Dave',

lastname: 'Davidds',

year: 1990,

},

];

const App = () => <List *list*={list} />;

If you would take this one step further, you could also extract the ListItem component into its own file and let the List component import the ListItem component:

- src/

--- App.js

--- List.js

--- ListItem.js

However, as said before, this may take it too far, because so far the ListItem component is tightly coupled to the List component. Hence it would be okay to leave it in the src/List.js file. I follow the rule of thumb that whenever a React component becomes a [reusable React component](https://www.robinwieruch.de/react-reusable-components), I split it out as a standalone file, like we did with the List component, to make it accessible for other React components.

## **FROM REACT FILES TO REACT FOLDERS**

From here it becomes more interesting and more opinionated. Every React component grows in complexity eventually. Not only because more logic is added (e.g. more JSX with [conditional rendering](https://www.robinwieruch.de/conditional-rendering-react) or [React Hooks](https://www.robinwieruch.de/react-hooks)), but also because there are more technical concerns like styling and tests. A naive approach would be to add more files next to each React component. For example, let's say every React component has a test and a style file:

- src/

--- App.js

--- App.test.js

--- App.css

--- List.js

--- List.test.js

--- List.css

One can already see that this doesn't scale well, because with every additional component in the src/ folder we will lose more sight of every individual component. That's why I like to have one folder for each React component:

- src/

--- App/

----- index.js

----- test.js

----- style.css

--- List/

----- index.js

----- test.js

----- style.css

The naming of these files is up to you. For example, index.js may become component.js or test.js may become spec.js. Moreover, if you are not using CSS but something like [Styled Components](https://www.robinwieruch.de/react-styled-components), your file extension may change from style.css to style.js too. Once you get used to your naming convention, you can just search for "List index" or "App test" in your IDE for opening each file. If you collapse all component folders, you have a very concise and clear folder structure:

- src/

--- App/

--- List/

If there are more technical concerns for a component, for example you may want to extract custom hooks into their own file for certain components, you can scale this approach horizontally within the component folder:

- src/

--- App/

----- index.js

----- test.js

----- style.css

--- List/

----- index.js

----- test.js

----- style.css

----- hooks.js

If you decide to keep your List/index.js more lightweight by extracting the ListItem component in its own file, then you may want to try the following folder structure:

- src/

--- App/

----- index.js

----- test.js

----- style.css

--- List/

----- index.js

----- test.js

----- style.css

----- ListItem.js

Here again you can go one step further by giving the component its own folder with all other technical concerns like tests and styles:

- src/

--- App/

----- index.js

----- test.js

----- style.css

--- List/

----- index.js

----- test.js

----- style.css

----- ListItem/

------- index.js

------- test.js

------- style.css

Important: From here on you need to be careful to not nest too deeply your components into each other. My rule of thumb is that I am never nesting components more than two levels, so the List and ListItem folders would be alright, but the ListItem's folder shouldn't have another nested folder.

After all, if you are not going beyond midsize React projects, this is in my opinion the way to go to structure your React components. However, as I mentioned, this is highly opinionated and may not meet everyones taste.

## **TECHNICAL FOLDER SEPARATION**

The next step will help you to structure midsize to large React applications. It separates features from components which are used by more than one component. Take the following folder structure as example:

- src/

--- components/

----- App/

------- index.js

------- test.js

------- style.css

----- List/

------- index.js

------- test.js

------- style.css

We group our previous React components into a new components/ folder. This gives us another vertical layer for creating folders for other categories. For example, at some point you may have React Hooks that can be used by more than one component. So instead of coupling the hook tightly to a component, you can put the implementation of it in a dedicated folder which can be used by all React components:

- src/

--- components/

----- App/

------- index.js

------- test.js

------- style.css

----- List/

------- index.js

------- test.js

------- style.css

--- hooks/

----- useClickOutside/

------- index.js

----- useData/

------- index.js

This doesn't mean that all hooks should end up in this folder though. React Hooks which are still only used by one component should remain in the component's file or maybe in a separate hooks.js file in the component's folder. Only hooks that can be consumed by all React components end up in the hooks/ folder.

The same strategy may apply if you are using [React Context](https://www.robinwieruch.de/react-context) in your project which needs to be accessed globally by all your other files:

- src/

--- components/

----- App/

------- index.js

------- test.js

------- style.css

----- List/

------- index.js

------- test.js

------- style.css

--- hooks/

----- useClickOutside/

------- index.js

----- useData/

------- index.js

--- context/

----- Session/

------- index.js

From here, there may be other utilities which need to be accessible to your components and hooks. For miscellaneous utilities I usually create a services/ folder. The name is up to you, but again it's the principal of making logic available to other code in our project which drives this technical separation:

- src/

--- components/

----- App/

------- index.js

------- test.js

------- style.css

----- List/

------- index.js

------- test.js

------- style.css

--- hooks/

----- useClickOutside/

------- index.js

----- useData/

------- index.js

--- context/

----- Session/

------- index.js

--- services/

----- ErrorTracking/

------- index.js

------- test.js

----- Format/

------- Date/

--------- index.js

--------- test.js

------- Currency/

--------- index.js

--------- test.js

Take for instance the Date/index.js file. The implementation details may look like the following:

export const formatDateTime = (date) =>

new Intl.DateTimeFormat('en-US', {

year: 'numeric',

month: 'numeric',

day: 'numeric',

hour: 'numeric',

minute: 'numeric',

second: 'numeric',

hour12: false,

}).format(date);

export const formatMonth = (date) =>

new Intl.DateTimeFormat('en-US', {

month: 'long',

}).format(date);

Fortunately [JavaScript's Intl API](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl) gives us excellent tools for date conversions. However, instead of using the API directly in my React components, I like to have a service for it, because only this way I can guarantee that my components have only a slim set of actively used date formatting options available for my application. Otherwise there would be lots of different date formats in a growing application.

Now it's possible to not only import each date formatting function individually:

import { formatMonth } from '../../services/format/date';

const month = formatMonth(new Date());

But also as a service, as an encapsulated module in other words, what I usually like to do:

import \* as dateService from '../../services/format/date';

const month = dateService.formatMonth(new Date());

It may become difficult to import things with relative paths now. Therefore I always would opt-in [Babel's Module Resolver](https://www.robinwieruch.de/babel-module-resolver) for aliases. Afterward, your import may look like the following:

import \* as dateService from '@format/date';

const month = dateService.formatMonth(new Date());

After all, I like this technical separation of concerns, because it gives every folder a dedicated purpose and it encourages sharing functionality across the application.

## **DOMAIN FOLDER SEPARATION**

The last step will help you to structure large React applications. This happens when you find yourself with lots of subfolders in your technical separated folders. The example doesn't show the full extent, but I hope you get the point:

- src/

--- components/

----- App/

----- List/

----- Input/

----- Button/

----- Checkbox/

----- Profile/

----- Avatar/

----- MessageItem/

----- MessageList/

----- PaymentForm/

----- PaymentWizard/

----- ErrorMessage/

----- ErrorBoundary/

From here I would use the components/ folder only for reusable components (e.g. UI components). Every other component should move to a domain centred folder. The names of the folders are again up to you:

- src/

--- domain/

----- User/

------- Profile/

------- Avatar/

----- Message/

------- MessageItem/

------- MessageList/

----- Payment/

------- PaymentForm/

------- PaymentWizard/

----- Error/

------- ErrorMessage/

------- ErrorBoundary/

--- components/

----- App/

----- List/

----- Input/

----- Button/

----- Checkbox/

If a PaymentForm needs access to a Button or Input, it imports it from the reusable UI components folder. If a MessageList component needs an abstract List component, it imports it as well. Furthermore, if a service from the previous step is tightly coupled to a domain, then move the service to the specific domain folder. The same may apply to other folders which were previously separated by technical concern:

- src/

--- domain/

----- User/

------- Profile/

------- Avatar/

----- Message/

------- MessageItem/

------- MessageList/

----- Payment/

------- PaymentForm/

------- PaymentWizard/

------- services/

--------- Currency/

----------- index.js

----------- test.js

----- Error/

------- ErrorMessage/

------- ErrorBoundary/

------- services/

--------- ErrorTracking/

----------- index.js

----------- test.js

--- components/

--- hooks/

--- context/

--- services/

----- Format/

------- Date/

--------- index.js

--------- test.js

Whether there should be an intermediate services/ folder in each domain folder is up to you. You could also leave out the folder and put the ErrorTracking/ folder directly into Error/. However, this may be confusing, because ErrorTracking should be marked somehow as a service and not as a React component. As alternative you could also use a [kebab-case naming convention, which is better than PascalCase anyway](https://www.robinwieruch.de/javascript-naming-conventions), by going with error-tracking/ instead of ErrorTracking/.

There is lots of room for your personal touch here. After all, this step is just about bringing the domains together which allows teams in your company to work on specific domains without having to touch files across the project.

Having all this written, I hope it helps one or the other person or team structuring their React project. Keep in mind that none of the shown approaches is set in stone. In contrast, I encourage you to apply your personal touch to it. Since every React project grows in size over time, most of the folder structures evolve very naturally as well. Hence the 5 step process to give you some guidance if things get out of hand.

## What Is CORS?

According to [developer.mozilla.org](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS):

**Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to let a user agent gain permission to access selected resources from a server on a different origin (domain) than the site currently in use.**

This allows you to access images, PDFs...heck, even other web pages to display in your own domain pages. That's what makes the web so...linkable.

While it's perfectly fine for images and other resource types, there's something to be said for Web API calls from JavaScript.

In an Intranet environment, I don't even see CORS as a thought when calling an internal Web API through JavaScript.

Images and similar documents, maybe, but Web APIs?

Nope. Not convinced yet.

## Why NOT Use CORS?

There are a couple of reasons why I have issues with making Web API calls inside a browser using JavaScript.

### Not Scalable

It's true the fastest way between two points is a straight line, but not when writing APIs.

I understand eliminating the middleman by not posting back to the server, but there's a problem with that type of thinking.

Writing JavaScript code to make a direct call to a Web API is a quick win, but if we can't reuse that code anywhere else, it's a one-hit wonder and isn't scalable.

It can only be used on that page. In the JavaScript code. Period.

### Security Issues

[Cross-site Scripting](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) (XSS) is a big thing lately and this type of code screams "security issue!"

Along with XSS, another issue could be redirects.

If your JavaScript has a callback operation and you don't have a security check for your domain, a well-versed developer can intercept the callback and redirect someone to another URL.

These are called [unvalidated callbacks](https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet) and you don't want these either.

I also strongly feel it unnecessary to even go into why it's bad practice for you to not only expose your API call in JavaScript but to expose your API keys and any other "encrypted" information for the world to see.

If you are a developer and feel this last statement is complete BS, you may want to take a security course. Just sayin'.

### Unnecessary Hacks

Over the years, I've seen too many developers dive into a CORS rabbit hole when it comes to public websites and they come out with various hacks...even to the point of modifying a Windows Registry on client machines to get their JavaScript to connect to a different server.

Nowadays, it's definitely easier, but some legacy applications I've seen (and even recent ones) have some scary hacks which make my hair even whiter than it currently is.

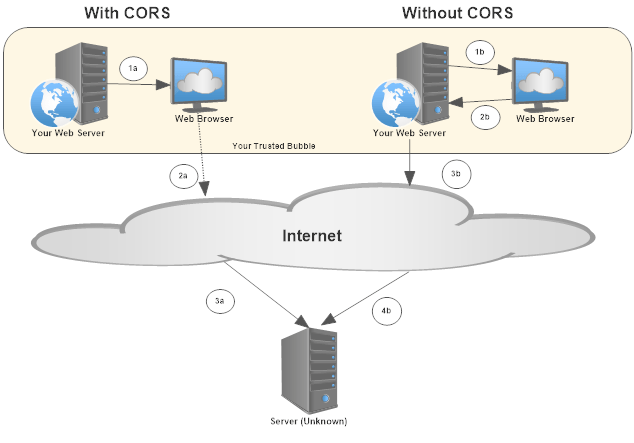
Yeah, not pretty.

## How Do We Solve This?

Most developers don't even think about CORS and just go about their business, but there are some administrators who lock down resources on a server and that's when developers go off the deep end and try to figure out ways around it.

This is what gives us bad practices for accessing resources.

After our review from above, developers have two choices.



### With CORS

1. A request for a web page returns the HTML, CSS, and JavaScript to your web browser.
2. After an action, your JavaScript makes a call.
3. The server (known to you at design-time which, by the way, you don't control) receives the request and returns data (JSON, XML, or even HTML).

Let me be clear about this option: **THIS IS NOT AN OPTION!**

After reviewing the drawbacks from above, the best approach to use a Web API is to call a familiar face: your own server. You received your HTML page from a trusted source (your server, duh!) so why not just call the server and let the server handle the grunt work?

Let's look at the option on the right side.

### Without CORS

1. A request for a web page returns the HTML, CSS, and JavaScript to your web browser.
2. After an action, your JavaScript calls a POST to your server.
3. Your server makes the API call instead of the browser making the call.
4. Your request makes it to the server (again, a server you don't control) and responds to your request.

Once your server receives the data, you can format it the way you want and return the data to the browser.

Do you know how many issues you can solve by letting your server perform the API calls?

So, yes, Plan B is probably your best bet.

## Conclusion

With the rise of Web APIs over the last decade, I can see why developers would want to just "make the call" and be done with it.

Heck, I've even seen someone create a SOAP request in JavaScript to make a call from the browser when it could've been done simply on the server in less than 10 lines of code.

However, based on everything I've seen with CORS, it makes sense to simply "play it safe" by calling my web server and use the server-side language (C# in our cases) to make all of the hard decisions (web API is down, wrong data was returned, etc.). Placing all of that inside JavaScript is a bit too much, me thinks.

Placing your Web API calls on the server instead of in your client-side JavaScript provides better scalability, security, and maintenance in the long run.

# Keeping credentials secure when making API calls with JavaScript

To authenticate you, the API may require:

1. Your username and password.
2. A *key* and *secret*.
3. An *API key* or *OAuth token*.

These are often passed to the API as query string values on the endpoint URL. For example, here’s how you make a call to the New York Times API.

https://api.nytimes.com/svc/topstories/v2/technology.json?api-key=my\_api\_key\_1234

**Including authentication credentials in your JavaScript is bad**[**#**](https://gomakethings.com/keeping-credentials-secure-when-making-api-calls-with-javascript/#including-authentication-credentials-in-your-javascript-is-bad)

The big challenge with using APIs that require authentication in your JavaScript is that you’re forced to expose your API credentials to use them.

Anyone who knows how to view source or view requests in their browser’s Developer Tools can view those credentials, steal them, and use them to access the API as you.

For APIs that let you send new data or update and delete existing data, that can be really dangerous. It’s also an issue for APIs that expose private data, restrict the number of calls you can make, or cost money to use.

**As a rule of thumb, *never* include credentials in client side JavaScript.**

The only time I personally would make an exception to that are for APIs that are:

1. Free, and
2. Only allow GET requests, and
3. Surface public data that’s accessible elsewhere.

Bonus points of the credentials are restricted in use to a specific domain or URL.

**How to securely use authenticated APIs in JavaScript**[**#**](https://gomakethings.com/keeping-credentials-secure-when-making-api-calls-with-javascript/#how-to-securely-use-authenticated-apis-in-javascript)

What if you need or want to an API that doesn’t mean those criteria? How can you access that data with JavaScript if you can’t include your credentials in your JS?

The trick is to setup an API endpoint on a server that you can call with your JavaScript.

This middleware API stores your credentials securely on the server, and makes the real API call on your request. It then sends back the data, optionally filtering out any data you don’t want exposed publicly first.

**AN EXAMPLE: THE MAILCHIMP API**[**#**](https://gomakethings.com/keeping-credentials-secure-when-making-api-calls-with-javascript/#an-example-the-mailchimp-api)

For example, [the Mailchimp API](https://developer.mailchimp.com/) requires an API key and private list ID to subscribe users.

I don’t want to expose either of those publicly, or someone could spam subscribe my list, delete subscribers, and so on.

I created [a WordPress plugin](https://github.com/cferdinandi/gmt-mailchimp-wp-rest-api) that uses the newish [WP REST API](https://developer.wordpress.org/rest-api/) to create a custom endpoint I can call from my JavaScript.

http://my-site-url.com/wp-json/gmt-mailchimp/v1/subscribe/

In WordPress, I can save my API key and list ID. I can also add a list of allowed domains, and any requests that come from domains other than those are ignored.

On the server, it uses [the wp\_remote\_request() method](https://developer.wordpress.org/reference/functions/wp_remote_request/) and the arguments I send along to ping the actual Mailchimp API or subscribe a new user (or update an existing one). Then, it returns a status code letting me know if it worked or not.

This let’s me keep my authentication credentials secret and still use APIs in my JavaScript.

Which of the following is NOT one of the benefits of creating the middleware API as described in the article for making authenticated API calls in JavaScript?

The idea behind this approach is that your JavaScript code will call the middleware API, which will then call the authenticated API. While your JavaScript is not directly calling the authenticated API, this method is more secure because your credentials won’t be exposed on the client-side. It allow you to securely make calls directly authenticated API in a single step.

# Chapter: 33 Introduction: Adding a PostgreSQL Database

# **What is an ORM and Why You Should Use it**

Before we talk about what an Object-Relational-Mapper is, it might be better to talk about [Object-Relational-Mapping](https://en.wikipedia.org/wiki/Object-relational_mapping) as a concept first.

Unless you’ve worked exclusively with NoSQL databases, you’ve likely written your fair share of SQL queries. They usually look something like this:

SELECT \* FROM users WHERE email = 'test@test.com';

Object-relational-mapping is the idea of being able to write queries like the one above, as well as much more complicated ones, using the object-oriented paradigm of your preferred programming language.

Long story short, we are trying to interact with our database using our language of choice *instead* of SQL.

Here’s where the Object-relational-**mapper** comes in. When most people say “ORM” they are referring to a library that implements this technique. For example, the above query would now look something like this:

var orm = require('generic-orm-libarry');var user = orm("users").where({ email: 'test@test.com' });

As you can see, we are using an imaginary ORM library to execute the exact same query, except we can write it in JavaScript (or whatever language you’re using). We can use the same languages we know and love, and also abstract away some of the complexity of interfacing with a database.

As with any technique, there are tradeoffs that should be considered when using an ORM.

Let’s take a look at some of the pros and cons!

# **What are some pros of using an ORM? 👍**

1. You get to write in the language you are already using anyway. If we’re being honest, we probably aren’t the greatest at writing SQL statements. SQL is a ridiculously powerful language, but most of us don’t write in it often. We do, however, tend to be much more fluent in one language or another and being able to leverage that fluency is awesome!
2. It abstracts away the database system so that switching from MySQL to PostgreSQL, or whatever flavor you prefer, is easy-peasy.
3. Depending on the ORM you get a lot of advanced features out of the box, such as support for transactions, connection pooling, migrations, seeds, streams, and all sorts of other goodies.
4. Many of the queries you write will perform better than if you wrote them yourself.

# **What are some cons of using an ORM? 👎**

1. If you are a master at SQL, you can probably get more performant queries by writing them yourself.
2. There is overhead involved in learning how to use any given ORM.
3. The initial configuration of an ORM can be a headache.
4. As a developer, it is important to understand what is happening under the hood. Since ORMs can serve as a crutch to avoid understanding databases and SQL, it can make you a weaker developer in that portion of the stack.

# **What are some popular ORMs?**

[Wikipedia has a great list of ORMs that exist for just about any language](https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software). That list is missing JavaScript, which is my language of choice, so I will throw my hat in the ring for [Knex.js](https://knexjs.org/#Interfaces-Streams).

They’re not paying me to say that, I’ve simply enjoyed working with their software and I don’t have any experience with other JavaScript ORMs. [This article](https://medium.freecodecamp.org/a-comparison-of-the-top-orms-for-2018-19c4feeaa5f) might provide more insightful feedback for JavaScript specifically.

# Mockaroo

Mockaroo is an online data generator for generating realistic test data in CSV, JSON, SQL, or Excel format. This is helpful if you want some mock data to test your application.

# Chapter: 34 Introduction: Security, Authentication, and Authorization

The Internet is a dangerous place! With great regularity, we hear about websites becoming unavailable due to denial of service attacks, or displaying modified (and often damaging) information on their homepages. In other high-profile cases, millions of passwords, email addresses, and credit card details have been leaked into the public domain, exposing website users to both personal embarrassment and financial risk.

The purpose of website security is to prevent these (or any) sorts of attacks. The more formal definition of website security is the act/practice of protecting websites from unauthorized access, use, modification, destruction, or disruption.

Effective website security requires design effort across the whole of the website: in your web application, the configuration of the web server, your policies for creating and renewing passwords, and the client-side code. While all that sounds very ominous, the good news is that if you're using a server-side web framework, it will almost certainly enable "by default" robust and well-thought-out defense mechanisms against a number of the more common attacks. Other attacks can be mitigated through your web server configuration, for example by enabling HTTPS. Finally, there are publicly available vulnerability scanner tools that can help you find out if you've made any obvious mistakes.

The rest of this article gives you more details about a few common threats and some of the simple steps you can take to protect your site.

**Note**: This is an introductory topic, designed to help you start thinking about website security, but it is not exhaustive.

## [Website security threats](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security#website_security_threats)

This section lists just a few of the most common website threats and how they are mitigated. As you read, note how threats are most successful when the web application either trusts, or is not paranoid enough about the data coming from the browser.

### [Cross-Site Scripting (XSS)](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security#cross-site_scripting_xss)

XSS is a term used to describe a class of attacks that allow an attacker to inject client-side scripts through the website into the browsers of other users. Because the injected code comes to the browser from the site, the code is trusted and can do things like send the user's site authorization cookie to the attacker. When the attacker has the cookie, they can log into a site as though they were the user and do anything the user can, such as access their credit card details, see contact details, or change passwords.

**Note**: XSS vulnerabilities have been historically more common than any other type of security threat.

The XSS vulnerabilities are divided into reflected and persistent, based on how the site returns the injected scripts to a browser.

* A reflected XSS vulnerability occurs when user content that is passed to the server is returned immediately and unmodified for display in the browser. Any scripts in the original user content will be run when the new page is loaded.  
  For example, consider a site search function where the search terms are encoded as URL parameters, and these terms are displayed along with the results. An attacker can construct a search link that contains a malicious script as a parameter (e.g., http://mysite.com?q=beer<script%20src="http://evilsite.com/tricky.js"></script>) and email it to another user. If the target user clicks this "interesting link", the script will be executed when the search results are displayed. As discussed earlier, this gives the attacker all the information they need to enter the site as the target user, potentially making purchases as the user or sharing their contact information.
* A persistent XSS vulnerability occurs when the malicious script is stored on the website and then later redisplayed unmodified for other users to execute unwittingly.  
  For example, a discussion board that accepts comments that contain unmodified HTML could store a malicious script from an attacker. When the comments are displayed, the script is executed and can send to the attacker the information required to access the user's account. This sort of attack is extremely popular and powerful, because the attacker might not even have any direct engagement with the victims.

While the data from POST or GET requests is the most common source of XSS vulnerabilities, any data from the browser is potentially vulnerable, such as cookie data rendered by the browser, or user files that are uploaded and displayed.

The best defense against XSS vulnerabilities is to remove or disable any markup that can potentially contain instructions to run the code. For HTML this includes elements, such as <script>, <object>, <embed>, and <link>.

The process of modifying user data so that it can't be used to run scripts or otherwise affect the execution of server code is known as input sanitization. Many web frameworks automatically sanitize user input from HTML forms by default.

### [SQL injection](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security#sql_injection)

SQL injection vulnerabilities enable malicious users to execute arbitrary SQL code on a database, allowing data to be accessed, modified, or deleted irrespective of the user's permissions. A successful injection attack might spoof identities, create new identities with administration rights, access all data on the server, or destroy/modify the data to make it unusable.

SQL injection types include Error-based SQL injection, SQL injection based on boolean errors, and Time-based SQL injection.

This vulnerability is present if user input that is passed to an underlying SQL statement can change the meaning of the statement. For example, the following code is intended to list all users with a particular name (userName) that has been supplied from an HTML form:

statement = "SELECT \* FROM users WHERE name = '" + userName + "';"

Copy to Clipboard

If the user specifies a real name, the statement will work as intended. However, a malicious user could completely change the behavior of this SQL statement to the new statement in the following example, by specifying the text in bold for the userName.

SELECT \* FROM users WHERE name = 'a';DROP TABLE users; SELECT \* FROM userinfo WHERE 't' = 't';

Copy to Clipboard

The modified statement creates a valid SQL statement that deletes the users table and selects all data from the userinfo table (which reveals the information of every user). This works because the first part of the injected text (a';) completes the original statement.

To avoid this sort of attack, you must ensure that any user data that is passed to an SQL query cannot change the nature of the query. One way to do this is to [escape](https://en.wikipedia.org/wiki/Escape_character) all the characters in the user input that have a special meaning in SQL.

**Note**: The SQL statement treats the **'** character as the beginning and end of a string literal. By putting a backslash in front of this character (**\'**), we escape the symbol, and tell SQL to instead treat it as a character (just a part of the string).

In the following statement, we escape the **'** character. The SQL will now interpret the name as the whole string in bold (which is a very odd name indeed, but not harmful).

SELECT \* FROM users WHERE name = 'a\';DROP TABLE users; SELECT \* FROM userinfo WHERE \'t\' = \'t';

Copy to Clipboard

Web frameworks will often take care of the character escaping for you. Django, for example, ensures that any user-data passed to querysets (model queries) is escaped.

**Note**: This section draws heavily on the information in [Wikipedia here](https://en.wikipedia.org/wiki/SQL_injection).

### [Cross-Site Request Forgery (CSRF)](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security#cross-site_request_forgery_csrf)

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user’s knowledge or consent.

This type of attack is best explained by example. John is a malicious user who knows that a particular site allows logged-in users to send money to a specified account using an HTTP POST request that includes the account name and an amount of money. John constructs a form that includes his bank details and an amount of money as hidden fields, and emails it to other site users (with the Submit button disguised as a link to a "get rich quick" site).

If a user clicks the submit button, an HTTP POST request will be sent to the server containing the transaction details and any client-side cookies that the browser associated with the site (adding associated site cookies to requests is normal browser behavior). The server will check the cookies, and use them to determine whether or not the user is logged in and has permission to make the transaction.

The result is that any user who clicks the Submit button while they are logged in to the trading site will make the transaction. John gets rich.

**Note**: The trick here is that John doesn't need to have access to the user's cookies (or access credentials). The browser of the user stores this information and automatically includes it in all requests to the associated server.

One way to prevent this type of attack is for the server to require that POST requests include a user-specific site-generated secret. The secret would be supplied by the server when sending the web form used to make transfers. This approach prevents John from creating his own form, because he would have to know the secret that the server is providing for the user. Even if he found out the secret and created a form for a particular user, he would no longer be able to use that same form to attack every user.

Web frameworks often include such CSRF prevention mechanisms.

### [Other threats](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security#other_threats)

Other common attacks/vulnerabilities include:

* [Clickjacking](https://www.owasp.org/index.php/Clickjacking). In this attack, a malicious user hijacks clicks meant for a visible top-level site and routes them to a hidden page beneath. This technique might be used, for example, to display a legitimate bank site but capture the login credentials into an invisible [<iframe>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe) controlled by the attacker. Clickjacking could also be used to get the user to click a button on a visible site, but in doing so actually unwittingly click a completely different button. As a defense, your site can prevent itself from being embedded in an iframe in another site by setting the appropriate HTTP headers.
* [Denial of Service](https://developer.mozilla.org/en-US/docs/Glossary/Distributed_Denial_of_Service) (DoS). DoS is usually achieved by flooding a target site with fake requests so that access to a site is disrupted for legitimate users. The requests may be numerous, or they may individually consume large amounts of resource (e.g., slow reads or uploading of large files). DoS defenses usually work by identifying and blocking "bad" traffic while allowing legitimate messages through. These defenses are typically located before or in the web server (they are not part of the web application itself).
* [Directory Traversal](https://en.wikipedia.org/wiki/Directory_traversal_attack) (File and disclosure). In this attack, a malicious user attempts to access parts of the web server file system that they should not be able to access. This vulnerability occurs when the user is able to pass filenames that include file system navigation characters (for example, ../../). The solution is to sanitize input before using it.
* [File Inclusion](https://en.wikipedia.org/wiki/File_inclusion_vulnerability). In this attack, a user is able to specify an "unintended" file for display or execution in data passed to the server. When loaded, this file might be executed on the web server or the client-side (leading to an XSS attack). The solution is to sanitize input before using it.
* [Command Injection](https://www.owasp.org/index.php/Command_Injection). Command injection attacks allow a malicious user to execute arbitrary system commands on the host operating system. The solution is to sanitize user input before it might be used in system calls.

For a comprehensive listing of website security threats see [Category: Web security exploits](https://en.wikipedia.org/wiki/Category:Web_security_exploits) (Wikipedia) and [Category: Attack](https://www.owasp.org/index.php/Category:Attack) (Open Web Application Security Project).

## [A few key messages](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security#a_few_key_messages)

Almost all of the security exploits in the previous sections are successful when the web application trusts data from the browser. Whatever else you do to improve the security of your website, you should sanitize all user-originating data before it is displayed in the browser, used in SQL queries, or passed to an operating system or file system call.

Important: The single most important lesson you can learn about website security is to **never trust data from the browser**. This includes, but is not limited to data in URL parameters of GET requests, POST requests, HTTP headers and cookies, and user-uploaded files. Always check and sanitize all incoming data. Always assume the worst.

A number of other concrete steps you can take are:

* Use more effective password management. Encourage strong passwords. Consider two-factor authentication for your site, so that in addition to a password the user must enter another authentication code (usually one that is delivered via some physical hardware that only the user will have, such as a code in an SMS sent to their phone).
* Configure your web server to use [HTTPS](https://developer.mozilla.org/en-US/docs/Glossary/https) and [HTTP Strict Transport Security](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security) (HSTS). HTTPS encrypts data sent between your client and server. This ensures that login credentials, cookies, POST requests data and header information are not easily available to attackers.
* Keep track of the most popular threats (the [current OWASP list is here](https://owasp.org/www-project-top-ten/)) and address the most common vulnerabilities first.
* Use [vulnerability scanning tools](https://owasp.org/www-community/Vulnerability_Scanning_Tools) to perform automated security testing on your site. Later on, your very successful website may also find bugs by offering a bug bounty [like Mozilla does here](https://www.mozilla.org/en-US/security/bug-bounty/faq-webapp/).
* Only store and display data that you need. For example, if your users must store sensitive information like credit card details, only display enough of the card number that it can be identified by the user, and not enough that it can be copied by an attacker and used on another site. The most common pattern at this time is to only display the last 4 digits of a credit card number.

Web frameworks can help mitigate many of the more common vulnerabilities.

Which of the following types of data poses the LEAST risk to the security of a website?

Most security threats come from user-originating data coming from the browser. Static files serverd from the web server.

Most security threats come from user-originating data coming from the browser.

XSS attacks allow an attacker to inject client-side scripts, such as through URL parameters, into the browsers of other users. Cross site scripting XXS

Grap from chapter 34 to chapter number 37

# CHAPTER: 38 Why Data Structures?

Data structures are the way we are able to store and retrieve data. The data structures that exist in programming languages are pretty similar to real-world systems that we use outside of the digital sphere. Imagine that you go to the grocery store. At this particular grocery store, the frozen pizza is stored next to the bell peppers and the toothbrushes are next to the milk. The store does not have signs that indicate where different items are located. In this disorganized grocery store, you would have a pretty difficult time trying to find what you were looking for!

Fortunately, most grocery stores have a clear order to the way the store is stocked and laid out. Similarly, data structures provide us with a way to organize information (including other data structures!) in a digital space.

**How are data structures used?**

Data structures handle four main functions for us:

* Inputting information
* Processing information
* Maintaining information
* Retrieving information

Inputting is largely concerned with how the data is received. What kind of information can be included? Will the new data be added to the beginning, end, or somewhere in the middle of the existing data? Does an existing point of data need to be updated or destroyed?

Processing gets at the way that data is manipulated in the data structure. This can occur concurrently or as a result of other processes that data structures handle. How does existing data that has been stored need to change to accommodate new, updated, or removed data?

Maintaining is focused on how the data is organized within the structure. Which relationships need to be maintained between pieces of data? How much memory must the system reserve (*allocate*) to accommodate the data?

Retrieving is devoted to finding and returning the data that is stored in the structure. How can we access that information again? What steps does the data structure need to take to get the information back to us?

Different types and use cases for data will be better suited to different manners of inputting, processing, storing, and retrieving. This is why we have several data structures to choose from… and the ability to create our own!

**Choosing the best data structure**

Your plumber probably would not be the best professional to diagnose an illness and your doctor probably wouldn’t be the best person to do your taxes — they are each better suited for other tasks! Similarly, different data structures are better suited for different tasks. Choosing the wrong data structure can result in slow or unresponsive code (and mess up your program!), so it’s important to consider a few factors as you make your decision:

1. What is the intended purpose for the data? Do any data structures have built-in functionality that is ideally suited for this purpose? Do you want to search, sort, or iterate data in a way in which certain data structures would be better suited than others?
2. Do you want or need control over how memory is set aside to store your data? Data structures that use *static memory allocation* (e.g., stacks or arrays) will manage memory for you and assume a fixed amount of memory upon instantiation with a cap on how much data may be added. Data structures that utilize *dynamic memory allocation* (e.g., heaps or linked lists) allow you to allocate and reallocate memory within the life of the program. While memory allocation is not something that you’ll need to consider in languages like Python or Javascript (these languages will manage memory for you, regardless of which data structure you use), it is something to bear in mind when working in other languages like C.
3. How long will it take different data structures to accomplish various tasks relative to other data structures? Technically, two data structures may both be able to accomplish the same task for you, but one may be quite a bit faster. This consideration, known as *runtime* will be covered further in depth when you explore all the nifty tricks of asymptotic notation.

As you’ve seen, data structures are the essential building blocks that we use to organize all of our digital information. Choosing the right data structure allows us to use the algorithms we want and keeps our code running smoothly. Understanding data structures and how to use them well can play a vital role in many situations including:

* technical interviews in which you may be asked to evaluate and determine runtime for data structures given specific algorithms
* day-to-day work for many software engineers who manipulate data stored in structures
* data science work where data is stored and accessed through data structures
* a whole lot more!

# Data Structure APIs

**A brief overview of APIs as they relate to JavaScript data structures.**

Data structures are all about choosing the right tool for the job.Consider the **Array** in JavaScript. It’s a really great data structure for storing ordered data because you can retrieve elements by index number. If you want the first element of an array, all you need to do is fetch it with index 0: **arrayName[0]**. It also provides all sorts of helpful methods for manipulating elements, such as **.push()**, **.pop()**, **.sort()**, and more. However, if you want to find out if a particular element exists in an array, you may need to iterate through the entire array.

What if I asked you to keep track of a series of numbers as I gave them to you, and then asked at the end whether I’d given you a particular number, you could probably do that in your memory. But if I asked you to do that in a computer program, you’d have to make choices about how to store the data. Let’s look at two possibilities of how we’d build **storeNumber()** and **doYouHaveThisNumber()** functions. Given the following list of numbers:

1, 250, -42, 0.4, 17

How might you store these numbers if I gave you each at a time? You might use an array:

const listOfNumbers = [];  
   
const storeNumber = num => listOfNumbers.push(num);  
   
const doYouHaveThisNumber = num => listOfNumbers.includes(num);

In this program, **storeNumber()** adds a number to the array, and **doYouHaveThisNumber()** returns **true** if that number exists in the array, and **false** otherwise. Looks pretty good, but what if you had 10000000 numbers? **doYouHaveThisNumber()** might start getting pretty slow, since **[Array.prototype.includes()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array" \t "_blank)** iterates through the entire array until it finds the input value.

Let’s try using another built-in data type in JavaScript, the **Object**. Since all we want to keep track of is whether we received a particular number, we can just store those numbers in an object, and set their values to **true** if we received them:

const receivedNumbers = {};  
   
const storeNumber = num => receivedNumbers[num] = true;  
   
const doYouHaveThisNumber = num => receivedNumbers[num] === true;

In this case, we’ll have the same result on the outside, but because retrieving a value from an object is much faster than iterating through an array, the overall result will be faster.

In both cases, the public API of the code, meaning the parts of the code that we want the end-user to interact with, remained the same: we had two functions, **storeNumber()** and **doYouHaveThisNumber()**. The underlying implementation, or the way the functionality was actually achieved, is what altered.

### What is an API?

API is an acronym for application programming interface. An API allows end-users to access properties and methods of data structures easily and without needing to do the “behind the scenes” work.

For example, if you want to add a new element to the end of an array, you don’t need to loop through the entire array, counting how many elements there are, and then setting **myArray[currentCount + 1]** equal to the new value. Instead, you can just call **.push()** with the value you want to add. As a JavaScript programmer, you don’t actually need to know the actual strategy, or the underlying implementation, of how **.push()** added an element to the end of the array in order to use it.

The [API of arrays](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) provides lots of useful functionality, from adding and removing elements to the start and end of the array, to iterator methods that call a function on each element. If you wanted to find the smallest number in an array of numbers, however, you’d have to implement that functionality yourself.

### Creating Your Own APIs

As you build your own data structures, you will implement the functionality to create public APIs. As in the example of **storeNumber()** and **doYouHaveThisNumber()**, the same public API can be implemented in different ways, so it’s important to think about the advantages and disadvantages of different implementations.

An API is like a message to end-users. Some languages have classes that can have methods or fields that are either public (can be called from anywhere) or private (can only be called from within the class). Public methods are the ones that end-users of that class can call, and private methods are only used by the class itself. JavaScript doesn’t really support this concept, so properties that aren’t meant to be public are often preceded by an underscore **\_**. Let’s look at an example where we want to build a data structure with a restricted API.

A stack is a data structure that only allows data to be added (pushed) or removed (popped) from the “top” of the stack. It just so happens that we could use an array as a stack, since it already has a **.push()** and **.pop()** method! However, arrays also allow you to add elements to the beginning or randomly access elements by index.

We’re not going to cover all the ins and outs of the stack data structure right now, but to demonstrate public API vs implementation, let’s build a quick custom **Stack** class:

class Stack {  
  constructor() {  
    this.\_array = [];  
  }  
}

In **Stack**, the array itself is stored as **\_array**, so it’s a signal to other developers that to use the **Stack** as intended, they shouldn’t need to access it directly. From there, we can implement the **.push()** and **.pop()** methods:

class Stack {  
  constructor() {  
    this.\_array = [];  
  }  
   
  push(newValue) {  
    this.\_array.push(newValue);  
  }  
   
  pop() {  
    return this.\_array.pop();  
  }  
}

Now we’ve created a **Stack** data structure that limits direct interaction with the underlying data to **.push()** and **.pop()**. A developer could still access our underlying array to do other manipulation:

const stack = new Stack();  
stack.\_array.unshift('value');

but they would then be breaking the intended behavior of the **Stack** class. The whole point of a public API is that we offer functionality to other end-users. If somebody were using our **Stack** class in a program, we could totally change the underlying implementation, and as long as the end-user API remained the same, their program should continue to function.

As you build your own classes and data structures, it’s important to keep in mind this distinction between implementation (what does this need internally to do its job) and the outside API (how should users of this actually interact with it?).

NODES: Nodes Introduction

Nodes are the fundamental building blocks of many computer science data structures. They form the basis for linked lists, stacks, queues, trees, and more.

An individual node contains data and links to other nodes. Each data structure adds additional constraints or behavior to these features to create the desired structure.

Consider the node depicted in the pane to the right. This node (node\_a) contains a piece of data (the number 5) and a link to another node (node\_b).

EXAMPLE OF NODES:

the node (node\_a) contains a piece of data(number 5) and link to another node(node\_b). node\_a

Link: node\_b

Data:5

**NODES: CONCEPTUAL**

Nodes Detail

The data contained within a node can be a variety of types, depending on the language you are using. In the previous example, it was an integer (the number 5), but it could be a string ("five"), decimal (5.1), an array ([5,3,4]) or nothing (null).

The link or links within the node are sometimes referred to as *pointers*. This is because they “point” to another node.

Typically, data structures implement nodes with one or more links. If these links are null, it denotes that you have reached the end of the particular node or link path you were previously following.

A variety of node implementations are depicted in the diagram. Examine the types of data and how some of the nodes are linked.

Diagram: on notebook:

Node Linking

Often, due to the data structure, nodes may only be linked to from a single other node. This makes it very important to consider how you implement modifying or removing nodes from a data structure.

If you inadvertently remove the single link to a node, that node’s data and any linked nodes could be “lost” to your application. When this happens to a node, it is called an orphaned node.

Examine the nodes in the diagram. node\_c is only linked to by node\_b. If you would like to remove node\_b but not node\_c, you can’t simply delete the link from node\_a to node\_b.

The most straightforward method to preserve node\_c would be to change the link in node\_a to point to node\_c instead of node\_b. However, some data structures may handle this in a different manner.

Nodes:

* Contain data, which can be a variety of data types
* Contain links to other nodes. If a node has no links, or they are all null, you have reached the end of the path you were following.
* Can be orphaned if there are no existing links to them

## Image

**1.**

Let’s begin by setting up the constructor for our Node class. Since nodes contain data, we want the constructor to expect a data argument of any type to be passed in. The constructor can assign the given argument to the data property of the Node instance.

Be sure to set the arguments to the appropriate properties in this class (i.e. this.data).

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Let’s check that our Node class has the correct data.

Underneath the Node class, instantiate a Node with any value and set it to firstNode. Then use console.log() to print out the instance’s data property. We should see the value that the node was instantiated with in the terminal.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

We also know that a node is aware of the other node it links to. We will represent this with the next property on the Node class.

Similar to how we created the data property in the constructor, let’s do the same with the next node property. When the node is first created, it is an orphan node (a node with no links). Set the next node to null.

Checkpoint 4 Passed

**4.**

Let’s check that next node property was successfully set in the constructor.

Underneath the node we previously created, use console.log() to print out the instance’s next node property. Check that null is output in the terminal.

Checkpoint

### Answer:

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

}

/\* instantiate creation

\*/

const firstNode = new Node('I am an instance of a Node!');

console.log(firstNode.data);

console.log(firstNode.next);

module.exports = Node;

Node Methods: Set Next Node

Currently when a node is created, the next node is initially set to null, and we do not have a formal way to change it. However, we want to allow the next node to be updated so it can be traversed and used in more complex data structures. For this, we will use a setter to modify this.next node property.

### Instructions

**1.**

Implement the .setNextNode() method in the Node class.

It should take node as an argument and update the next node property appropriately.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

To verify that our .setNextNode() performs as intended, let’s call the method on our Node instance. Create a second Node instance and set it to secondNode. Link it to the firstNode by passing secondNode to the call to setNextNode.

Now let’s print out firstNode so we can see it in its entirety using console.log(). We should see the second node instance set to the next node property instead of the default null value.

Checkpoint

### Answer:

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    this.next = node;

  }

}

const firstNode = new Node('I am an instance of a Node!');

const secondNode = new Node('I am the next Node!');

firstNode.setNextNode(secondNode);

console.log(firstNode);

module.exports = Node;

Node Methods: Set Next Node Validation

We arbitrarily set our next node to any argument that gets passed in. This can be problematic. Imagine if the next node accidentally gets set to a different data type, like a string. We run the risk of mistakenly confusing the string for a node, and we would have to build out special support to avoid traversing anything that is not a node.

To prevent these unnecessary complications, let’s add in a check that only allows arguments that are instanceof nodes or null. We want to allow null values as an argument in the event that we want to break the link between a node and its next node.

### Instructions

**1.**

Inside .setNextNode(), check if the node argument is an instanceof the Node class. If the argument is a Node or null, set this.next equal to node. Otherwise, throw an error.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

We know that we can set the next node to another node, so let’s verify that .setNextNode() will not accept an argument that is not a node.

Call the .setNextNode() method on our Node instance and pass it any argument that is not a node. We expect to see an error in the terminal because you didn’t set the next node to a Node instance.

Checkpoint 3 Passed

Node Methods: Get Next Node

We could continue accessing the next node property directly, like how we have been doing so far. However, if we ever want it to be given in a special way, we would want to use a getter to handle the preprocessing.

Let’s go ahead and create a simple .getNextNode() method that just returns the next node property.

### Instructions

**1.**

Implement the .getNextNode() method in the Node class.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Using the Node instance that we have already created, verify that the .getNextNode() method returns the correct next node property by logging the call from firstNode.

Checkpoint

### Answer:

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class.');

    }

  }

 /\* first answer\*/

  getNextNode() {

    return this.next;

  }

}

const firstNode = new Node('I am an instance of a Node!');

const secondNode = new Node('I am the next Node!');

firstNode.setNextNode(secondNode);

/\* 2nd answer\*/

console.log(firstNode.getNextNode());

module.exports = Node;

**1.**

Outside of Node, instantiate three more nodes.

* The first will represent our strawberry ice cream with the name, 'Berry Tasty'. Assign it to the variable, strawberryNode
* The second will represent our vanilla ice cream with the value, 'Vanilla'. Assign it to the variable, vanillaNode
* The third will represent our coconut ice cream with the value, 'Coconuts for Coconut'. Assign it to the variable, coconutNode

Checkpoint 2 Passed

**2.**

Now let’s put these ice cream nodes in order. vanilla goes first, followed by strawberry. Finally, coconut goes after strawberry.

Below the newly created nodes, use your .setNextNode() method so that:

* strawberryNode is the next node of vanillaNode
* coconutNode the next node of strawberryNode

Checkpoint 3 Passed

**3.**

Let’s walk through our ice cream nodes to make sure that they are linked in the correct order. Create a new currentNode and set it vanillaNode. We will use currentNode to iterate through the nodes, so declare it using let. Create a while loop that will only iterate when the currentNode is not null.

Inside the while loop, log out the currentNode’s data, and update currentNode to the next node.

We should see the ice cream flavors in order of vanilla, strawberry, and coconut in the terminal.

When your code is passing, take a moment to consider:

* What other ways do you think nodes could be helpful for keeping track of and storing information?
* What could happen if we added another link to the Node class?
* What if we created an instance of a Node with another Node instance?

Checkpoint

### Answer:

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class.');

    }

  }

  getNextNode() {

    return this.next;

  }

}

const strawberryNode = new Node('Berry Tasty');

const vanillaNode = new Node('Vanilla');

const coconutNode = new Node('Coconuts for Coconut');

vanillaNode.setNextNode(strawberryNode);

strawberryNode.setNextNode(coconutNode);

let currentNode = vanillaNode;

while(currentNode) {

  console.log(currentNode.data);

  currentNode = currentNode.getNextNode();

}

module.exports = Node;

**LINKED LISTS: CONCEPTUAL**

Linked List Introduction

Linked lists are one of the basic data structures used in computer science.

The head node is the node at the beginning of the list. Each node contains data and a link (or pointer) to the next node in the list. The list is terminated when a node’s link is null. This is called the tail node.

Consider a one-way air travel itinerary. The trip could involve traveling through several airports (nodes) connected by air travel segments (links). In this example, the initial departure city is the head node and the final arrival city is the tail node.

Since the nodes use links to denote the next node in the sequence, the nodes are not required to be sequentially located in memory. These links also allow for quick insertion and removal of nodes as you will see in future exercises.

Common operations on a linked list may include:

* adding nodes
* removing nodes
* finding a node
* traversing (or traveling through) the linked list

Linked List Example

* As an example, we added values to the linked list diagram from the introduction.
* This linked list contains three nodes (node\_a, node\_b, and node\_c).
* Each node in this particular list contains a string as its data. As the sequence is defined, the order is "cats", "dogs", and "birds".
* The list ends at node\_c, since the link within that node is set to null.

Linked Lists Adding and Removing Nodes

* With linked lists, because nodes are linked to from only one other node, you can’t just go adding and removing nodes willy-nilly without doing a bit of maintenance.

#### Adding a new node

* Adding a new node to the beginning of the list requires you to link your new node to the current head node. This way, you maintain your connection with the following nodes in the list.

#### Removing a node

* If you accidentally remove the single link to a node, that node’s data and any following nodes could be lost to your application, leaving you with orphaned nodes.
* To properly maintain the list when removing a node from the middle of a linked list, you need to be sure to adjust the link on the previous node so that it points to the following node.
* Depending on the language, nodes which are not referenced are removed automatically. “Removing” a node is equivalent to removing all references to the node.

Linked Lists:

* Are comprised of nodes
* The nodes contain a link to the next node (and also the previous node for bidirectional linked lists)
* Can be unidirectional or bidirectional
* Are a basic data structure, and form the basis for many other data structures
* Have a single head node, which serves as the first node in the list
* Require some maintenance in order to add or remove nodes
* The methods we used are an example and depend on the exact use case and/or programming language being used.

**LEARN LINKED LISTS: JAVASCRIPT**

Constructor and Adding to Head

Let’s implement a linked list in JavaScript. As you might recall, a linked list is a sequential chain of nodes. Remember that a node contains two elements:

* data
* a link to the next node

We are going to use a provided Node class, which you can find in **Node.js**. Make sure to use the proper Node methods throughout the lesson instead of accessing properties directly (ex. use someNode.getNextNode() instead of someNode.next).

Depending on the end-use of the linked list, there are a variety of methods that we can define. For our use, we want to be able to:

* add a new node to the beginning (head) of the list
* add a new node to the end (tail) of the list
* remove a node from the beginning (head) of the list
* print out the nodes in the list in order from head to tail

To start, we are going to create the LinkedList‘s constructor and .addToHead() method.

Ready? Let’s go!

**Instructions**

**1.**

The only property we need our linked list to have is a head, which we will add in our constructor. Inside the LinkedList class, define the constructor. It should take no parameters, and should set the list’s head to null.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Define an .addToHead() method that takes one parameter called data. Inside the method, create a Node const variable named newHead, and pass data as an argument.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Still inside your .addToHead() method, create a const variable named currentHead, and set it equal to the list’s head. Then change the list’s head to equal newHead.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Finally, still in your .addToHead() method, check if there is a current head to the list. If there is, set the list’s head’s next node to currentHead.

Answer:

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;/\* list head\*/

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;/\*head\*/

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

}

module.exports = LinkedList;

Adding to Tail

Now that we can add to the head of the linked list, the next step is to be able to add to the tail. This will require a few more steps since we don’t have a tail property in our linked list data structure.

To do this, we are going to start with a temporary tail variable that will be set equal to the list’s head. If there is no head, that means that the list is empty, and we will add the node to the head of the list. Otherwise, we will iterate through the list until we find the last node. Once we’ve found the current tail, we will add a pointer from that node to our new tail.

### Instructions

**1.**

Define an .addToTail() method for the LinkedList that has one parameter called data. Create a variable named tail, and set it equal to the list’s head. tail is going to change throughout the method, so make it a let variable.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Now that tail is equal to the head of the list, we want to check if it has any value. If tail has no value, then that means the list was empty, and we will be creating the head and tail with the data passed in.

To do this, check if tail has no value. If so, set the list’s head equal to a new Node that takes data as an argument.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

If tail does have a value, that means the list is not empty. In that case, we want to iterate through the list until we find the end, so we can add tail to the end of the list.

To do this, create an else after your if statement. Inside it, make a while loop that runs while tail has a next node. Inside the loop, set tail equal to its next node.

(If you accidentally create an infinite loop and your code won’t stop running, you can reload the page to stop it.)

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Finally, inside the same else block, but outside the while loop, set tail‘s next node equal to a new Node that takes data as an argument.

Checkpoint

Answer:

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

}

module.exports = LinkedList;

Removing the Head

So far we can:

* create a new LinkedList using its constructor
* add to the head of the list using .addToHead()
* add to the tail of the list using .addToTail()

Now we’re going to learn how to remove from the head of the list. To do this, we are first going to check to see if the list has a head. If it doesn’t, there is nothing to return. If there is a head, we will remove it by setting the list’s head equal to the original head’s next node, and then return that original head.

**Instructions**

**1.**

Define a .removeHead() method that takes no parameters. Inside the method, create a const variable named removedHead and set it equal to the list’s head. We will use this to keep track of the original head of the list.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

If removedHead has no value, return to end execution of the .removeHead() method.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside the if statement, set the list’s head equal to removedHead‘s next node.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Finally, return removedHead’s data.

Checkpoint

Answer:LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

}

module.exports = LinkedList;

NODE.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

Printing

Nice! Now we have a bunch of helpful LinkedList methods under our belt. Our next step is to create a .printList() method so we can see our list as it changes.

While it’s possible to just use console.log() on the list (try it!), we want to print it in a more understandable and readable way. console.log() will print the pointers of each node as well as the data, but we’re just going to print the data while maintaining the order of the list.

To do this, we will create a String that holds the data from every node in the list. We’ll start at the list’s head and iterate through the list, adding to our String as we go.

For example, if we had a list for the days of the week, starting with Sunday, .printList() would print it as follows:

<head> Sunday Monday Tuesday Wednesday Thursday Friday Saturday <tail>

**Instructions**

**1.**

Define a method named .printList(). Inside it, create:

* A let variable named currentNode, and set it equal to the list’s head
* Another let variable named output, and set it equal to '<head> '

Then, log output to the console

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

While currentNode doesn’t equal null, add its data and a ' ' to output. Then update currentNode to be its next node. Do this above your console.log() statement.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Finally, outside of the while loop, but before your console.log(), set output equal to itself concatenated with '<tail>'.

Checkpoint 4 Passed

Answer: LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

Using the Linked List

You finished your LinkedList class! Now we’re going to create an instance of that class and create a linked list of the seasons. We will add to it, remove from it, and finally print it out to check what we’ve done.

### Instructions

**1.**

In **seasons.js**, define a LinkedList named seasons. Print it out – what do you expect to see?

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Add 'summer' to the head of the seasons, then add 'spring' to the head. Print the list again.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Add 'fall' to the tail of seasons. Then add 'winter' to the tail and print the list again.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Finally, remove the element at the head of the list. Print the list to check that the correct element was removed.

Checkpoint 5 Passed

Answer: seasons.js

const LinkedList = require('./LinkedList');

const seasons = new LinkedList();

seasons.printList();

seasons.addToHead('summer');

seasons.addToHead('spring');

seasons.printList();

seasons.addToTail('fall');

seasons.addToTail('winter');

seasons.printList();

seasons.removeHead();

seasons.printList();

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    this.head = newHead;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let tail = this.head;

    if (!tail) {

      this.head = new Node(data);

    } else {

      while (tail.getNextNode() !== null) {

        tail = tail.getNextNode();

      }

      tail.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

We did this by:

* Using our Node class to hold the data and links between nodes
* Implementing a LinkedList class to handle external operations on the list, like adding and removing nodes
* Creating an instance of our list, and using our .printList() method to track the changes we made

# Swapping Elements in a Linked List

**Learn how to swap two nodes in a singly linked list in JavaScript.**

Since singly linked lists only have pointers from each node to its next node, swapping two nodes in the list isn’t as easy as doing so in an array (where you have access to the indices). You not only have to find the elements, but also reset the pointers around them to maintain the integrity of the list. This means keeping track of the two nodes to be swapped as well as the nodes preceding them.

Given an input of a linked list, **data1**, and **data2**, the general steps for doing so is as follows:

1. Iterate through the list looking for the node that matches **data1** to be swapped (**node1**), keeping track of the node’s previous node as you iterate (**node1Prev**)
2. Repeat step 1 looking for the node that matches **data2** (giving you **node2** and **node2Prev**)
3. If **node1Prev** is **null**, **node1** was the head of the list, so set the list’s head to **node2**
4. Otherwise, set **node1Prev**‘s next node to **node2**
5. If **node2Prev** is **null**, set the list’s head to **node1**
6. Otherwise, set **node2Prev**‘s next node to **node1**
7. Set **node1**‘s next node to **node2**‘s next node
8. Set **node2**‘s next node to **node1**‘s next node

### Finding the Matching and Preceding Nodes

Let’s look at what implementing steps 1 and 2 looks like. In order to swap the two nodes, we must first find them. We also need to keep track of the nodes that precede them so that we can properly reset their pointers. (We will use the **Node** class’s **.getNextNode()** method in order to access the next node.)

We will start by setting **node1** equal to the head of the list, and then creating a **while** loop that runs while **node1** isn’t **null**. Inside the loop, we will check if **node1**‘s data matches **data1**. If so, we **break** out of the loop as we have found the correct node. If there is no match, we update **node1Prev** to be **node1** and move **node1** to its next node:

function swapNodes(list, data1, data2) {  
  let node1 = list.head;  
  let node2 = list.head;  
  let node1Prev = null;  
  let node2Prev = null;  
   
  while (node1 !== null) {  
    if (node1.data === data1) {  
      break;  
    }  
    node1Prev = node1;  
    node1 = node1.getNextNode();  
  }  
}

At the end of this, we have found our matching node, and also saved its previous node, which we will use in the next step.

Fill in the Code

Fill in the code to complete the **while** loop that will find **node2** and **node2Prev**.

while (node2 !== null) {  
  if (node2.data === ) {  
    ;  
  }  
  node2Prev = ;  
  node2 = ;  
}

* data2
* break
* node2.getNextNode()
* node1Prev
* node1
* node2

Click or drag and drop to fill in the blank

Check Answer

### Updating the Preceding Nodes’ Pointers

Our next step is to set **node1Prev** and **node2Prev**‘s next nodes, starting with **node1Prev**. We will start by checking if **node1Prev** is **null**. If it is, then the **node1** is the head of the list, and so we will update the head to be **node2**. If **node1Prev** isn’t **null**, then we set its next node to **node2**:

// Still inside the swapNodes() function  
if (node1Prev === null) {  
  list.head = node2;  
} else {  
  node1Prev.setNextNode(node2);  
}

After this step, we have finished updating the pointers that point to our swapped nodes. The next step will be to update the pointers from them.

Fill in the Code

Fill in the code to update **node2Prev**‘s next node.

if (node2Prev === null) {  
  list.head = ;  
} else {  
  .setNextNode(node1);  
}

* node2Prev
* node1Prev
* node1
* node2

Click or drag and drop to fill in the blank

Check Answer

### Updating the Nodes’ Next Pointers

The last step is to update the pointers from **node1** and **node2**. This is relatively simple, and mirrors a swapping function for an array in that we will use a temporary variable.

Fill in the Code

Fill in the code to set **node1** and **node2**‘s next nodes.

let temp = node1.getNextNode();  
node1.setNextNode();  
node2.setNextNode();

* node1
* node1.getNextNode()
* node2
* temp
* node2.getNextNode()

Click or drag and drop to fill in the blank

Check Answer

### Edge Cases

We have completed the basic swap algorithm in JavaScript! However, we haven’t accounted for some edge cases. What if there is no matching node for one of the inputs? The current **swapNodes()** function will not run because we will try to access the next node of a node that is **null**. (Remember that our initial **while** loop only breaks if the matching node is found. Otherwise, it runs until the node is **null**.) Thankfully this has a quick fix. We can put in an **if** that checks if either **node1** or **node2** is **null**. If they are, we can print a statement that explains a match was not found, and **return** to end the method. We can put this right after the **while** loops that iterate through the list to find the matching nodes:

if (node1 === null || node2 === null) {  
  console.log('Swap not possible - one or more element is not in the list')  
  return;  
}

The last edge case is if the two nodes to be swapped are the same. While our current implementation will run without error, there’s no point in executing the whole function if it isn’t necessary. We can add a brief check at the beginning of the function that checks if the **data1** is the same as **data2**, and then **return** to end the function:

if (data1 === data2) {  
  console.log('Elements are the same - no swap needed.');  
  return;  
}

# Two-Pointer Linked List Techniques

**Learn how to approach Linked List problems with multiple iterator pointers.**

Many common singly linked list problems can be solved by iterating with two pointers. This is sometimes known as the runner technique.

## Two Pointers Moving in Parallel

Consider the following problem:

Create a function that returns the nth last element of a singly linked list.

In order to do this, you’ll need some way of knowing how far you are from the end of the list itself. However, in a singly linked list, there’s no easy way to iterate back through the list when you find the end.

If you want, you can try your hand at the problem directly, or we can walk through some approaches below.

### Approaches

One thing that might first come to mind is to use an array to store a representation of the linked list. While this approach results in an easy-to-read implementation, it could also use up lots of memory maintaining a dual representation of the same data. If the linked list has one million nodes, we’ll need one million pointers in an array to keep track of it! An approach like this results in an extra O(n) space being allocated.

const arrayNthLast = (list, n) => {  
  const linkedListArray = [];  
  let currentNode = list.removeHead();  
  while (currentNode) {  
    linkedListArray.push(currentNode);  
    currentNode = currentNode.getNextNode();  
  }  
  return list[list.length - n];  
}

Instead of creating an entire parallel list, we can solve this problem by using two pointers at different positions in the list but moving at the same rate. As in the previous example, we will use one pointer to iterate through the entire list, but we’ll also move a second pointer delayed **n** steps behind the first one.

**nthLastNodePointer = null**

**tailPointer = linked list head**

**count = 0**

**while tail pointer exists**

**move tail pointer forward**

**if count >= n**

**set nthLastNodePointer to head if it's still null or move it forward**

**increment count**

**return nthLastNodePointer**

### Implementation

Code Challenge

Complete the **nthLastNode()** function so that it returns the correct **Node** instance (or **null** if the **linkedList** is shorter than **n** elements). Do this without creating any additional new data structures (such as an array).

You can use the **testLinkedList** to experiment yourself. It contains a 50-length linked list with **data** values from **50 -> 49 -> ... 2 -> 1**

Run

Output:



Check Answer

### Solution

In JavaScript, we could implement the nth-last-node-finder function as such:

const nthLastNode = (linkedList, n) => {  
  let current = null;  
  let tailSeeker = linkedList.head;  
  let count = 0;  
  while (tailSeeker) {  
    tailSeeker = tailSeeker.next;  
    if (count >= n) {  
      if (!current) {  
        current = linkedList.head;  
      }  
      current = current.next;  
    }  
    count++  
  }  
  return current;  
}

The exact variable names aren’t important, and the internal implementation could be written in a number of ways, but the important part is that we are able to complete this problem efficiently–in O(n) time (we must iterate through the entire list once), and O(1) space complexity (we always use only three variables no matter what size the list is: two pointers and a counter).

## Pointers at Different Speeds

Another two-pointer technique involves sending pointers through the list at different iteration “speeds”.

Consider this problem:

Find the middle node of a linked list.

### Approaches

As before, it’s possible to find a solution by iterating through the entire list, creating an array representation, and then returning the middle index. But as before, this potentially takes up lots of extra space:

**create array**

**while the linked list has not been fully iterated through**

**push the current element onto array**

**move forward one node**

**return array[length / 2]**

Instead, we can use two pointers to move through the list. The first pointer takes two steps through the list for every one step that the second takes, so it iterates twice as fast.

**fastPointer = list head**

**slowPointer = list head**

**while fastPointer is not null**

**move fastPointer forward**

**if the end of the list has not been reached**

**move fastPointer forward again**

**move slowPointer forward**

**return slowPointer**

When the first pointer reaches the end of the list, the “slower” second pointer will be pointing to the middle element. Let’s visualize the steps of the algorithm:

**Starting State**

**F**

**S**

**1 2 3 4 5 6 7**

**First Tick**

**F**

**S**

**1 2 3 4 5 6 7**

**Second Tick**

**F**

**S**

**1 2 3 4 5 6 7**

**Third Tick**

**F**

**S**

**1 2 3 4 5 6 7**

**Final Tick**

**F**

**S**

**1 2 3 4 5 6 7 null**

As long as we always move the fast pointer first and check to see that it is not null before moving it and the slow pointer again, we’ll exit iteration at the proper time and have a reference to the middle node with the slow pointer.

# Doubly Linked Lists

Common operations on a doubly linked list may include:

* adding nodes to both ends of the list
* removing nodes from both ends of the list
* finding, and removing, a node from anywhere in the list
* traversing (or traveling through) the list

**DOUBLY LINKED LISTS: CONCEPTUAL**

Adding to the List

In a doubly linked list, adding to the list is a little complicated as we have to keep track of and set the node’s previous pointer as well as update the tail of the list if necessary.

**Adding to the Head**

When adding to the head of the doubly linked list, we first need to check if there is a current head to the list. If there isn’t, then the list is empty, and we can simply make our new node both the head and tail of the list and set both pointers to null. If the list is not empty, then we will:

* Set the current head’s previous pointer to our new head
* Set the new head’s next pointer to the current head
* Set the new head’s previous pointer to null

**Adding to the Tail**

Similarly, there are two cases when adding a node to the tail of a doubly linked list. If the list is empty, then we make the new node the head and tail of the list and set the pointers to null. If the list is not empty, then we:

* Set the current tail’s next pointer to the new tail
* Set the new tail’s previous pointer to the current tail
* Set the new tail’s next pointer to null
* **DOUBLY LINKED LISTS: CONCEPTUAL**
* Removing from the Head and Tail
* Due to the extra pointer and tail property, removing the head from a doubly linked list is slightly more complicated than removing the head from a singly linked list. However, the previous pointer and tail property make it much simpler to remove the tail of the list, as we don’t have to traverse the entire list to be able to do it.

#### Removing the Head

* Removing the head involves updating the pointer at the beginning of the list. We will set the previous pointer of the new head (the element directly after the current head) to null, and update the head property of the list. If the head was also the tail, the tail removal process will occur as well.

#### Removing the Tail

* Similarly, removing the tail involves updating the pointer at the end of the list. We will set the next pointer of the new tail (the element directly before the tail) to null, and update the tail property of the list. If the tail was also the head, the head removal process will occur as well.

## Image

Removing from the Middle of the List

It is also possible to remove a node from the middle of the list. Since that node is neither the head nor the tail of the list, there are two pointers that must be updated:

* We must set the removed node’s preceding node’s next pointer to its following node
* We must set the removed node’s following node’s previous pointer to its preceding node

There is no need to change the pointers of the removed node, as updating the pointers of its neighboring nodes will remove it from the list. If no nodes in the list are pointing to it, the node is orphaned.

Doubly Linked Lists Review

* Are comprised of nodes that contain links to the next and previous nodes
* Are bidirectional, meaning it can be traversed in both directions
* Have a pointer to a single head node, which serves as the first node in the list
* Have a pointer to a single tail node, which serves as the last node in the list
* Require the pointers at the head of the list to be updated after addition to or removal of the head
* Require the pointers at the tail of the list to be updated after addition to or removed of the tail
* Require the pointers of the surrounding nodes to be updated after removal from the middle of the list

Your browser history is another example of a doubly linked list. When you open your browser, the page that you land on is the head of your list. As you click on things and navigate to new pages, you are moving forward and adding to the tail of your list. If you ever want to go back to something you’ve already visited, you can use the “back” button to move backward through your list. Can you think of another computer use case for a doubly linked list?

**DOUBLY LINKED LISTS: JAVASCRIPT**

Node Class and Constructor

Let’s implement a doubly linked list in JavaScript. A doubly linked list is a sequential chain of nodes, just like a linked list. The nodes we used for our linked lists contained two elements:

* data
* a link to the next node

The difference between a doubly linked list and a linked list is that in a doubly linked list, there are pointers to the previous node as well as the next node. This means that the doubly linked list data structure has a tail property in addition to the head property that’s present in the linked list data structure.

Depending on the end-use of the doubly linked list, there are a variety of methods that we can define.

For our use, we want to be able to:

* add a new node to the beginning (head) of the list
* add a new node to the end (tail) of the list
* remove a node from the beginning (head) of the list
* remove a node from the end (tail) of the list
* find and remove a specific node by its data
* print out the nodes in the list in order from head to tail

We will reuse the .printList() method from our LinkedList class, but the rest will either be edited or new.

To start, we are going to look at the updated Node class and create the constructor.

Ready? Let’s get started!

**Instructions**

**1.**

We are going to use a provided Node class, which you can find in **Node.js**. We’ve added a previous property to the class, as well as .setPreviousNode() and .getPreviousNode() methods. Take a look at it to see the changes.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In your DoublyLinkedList class, create a constructor that has no parameters.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Inside your DoublyLinkedList constructor:

* Set the list’s head to null
* Set the list’s tail to null

Checkpoint 4 Passed

answer:DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  // Create your constructor below:

  constructor() {

    this.head = null;

    this.tail = null;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

Add To Head

In a linked list, we add to the head of the list by checking to see if there is already a head. We then either set the new node as the head (if there was no head) or update the head property, and link the past head to the new head.

Since a doubly linked list has an additional tail property and is built with nodes that each have two pointers, there are a few more steps:

* Start by checking to see if there is a current head to the list
* If there is (meaning the list is not empty), then we want to reset the pointers at the head of the list:
  + Set the current head’s previous node to the new head
  + Set the new head’s next node to the current head
* Update the head property to be the new head
* Finally, if there isn’t a current tail to the list (meaning the list was empty):
  + Update the tail property to be the new head since that node will be both the head and tail

**Instructions**

**1.**

Define an .addToHead() method that takes one parameter called data. Inside, create:

* A Node const variable named newHead that takes data as an argument
* A const variable named currentHead that’s set to the list’s head

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

If there is a current head to the list:

* Set currentHead‘s previous node to newHead
* Set newHead‘s next node to currentHead

Remember to use the Node class’s .setNextNode() and .setPreviousNode() methods.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside of the if, set the list’s head to the new head.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Lastly, if the list doesn’t have a tail, set the list’s tail to the new head.

Checkpoint 5 Passed

answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  // Create your .addToHead() method below:

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

Add To Tail

Since doubly linked lists have a tail property, we don’t have to iterate through the entire list to add to the tail like we did with a singly linked list. The new method will mirror what we did in our .addToHead() method:

* Start by checking to see if there is a current tail to the list
* If there is (meaning the list is not empty), then we want to reset the pointers at the tail of the list:
  + Set the current tail’s next node to the new tail
  + Set the new tail’s previous node to the current tail
* Update the tail property to be the new tail
* Finally, if there isn’t a current head to the list (meaning the list was empty):
  + Update the head property to be the new tail since that node will be both the head and tail

**Instructions**

**1.**

Define an .addToTail() method that takes one parameter called data. Inside, create:

* A Node const variable named newTail that takes data as an argument
* A const variable named currentTail, and set it equal to the list’s tail

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

If there is a current tail to the list:

* Set the current tail’s next node to newTail
* Set newTail‘s previous node to the current tail

Remember to use the Node class’s .setNextNode() and .setPreviousNode() methods.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside your if, set the list’s tail to the new tail.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Lastly, if the list doesn’t have a head, set the list’s head to the new tail.

Checkpoint 5 Passed

answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  // Create your .addToTail() method below:

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    this.head = newTail;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

Remove Head

Due to the added tail property, removing the head of the list in a doubly linked list is a little more complicated than doing so in a singly linked list:

* Start by checking if there’s a current head to the list. If there isn’t:
  + The list is empty, so there’s nothing to return, and the method ends
* Otherwise, update the list’s head to be the current head’s next node
* If there is still a head to the list (meaning the list had more than one element when we started):
  + Set the head’s previous node to null since there should be no node before the head of the list
* If the removed head was also the tail of the list (meaning there was only one element in the list), call .removeTail() to make the necessary changes to the tail of the list. (We will create this method in the next exercise!)
* Finally, return the old head

**Instructions**

**1.**

Define a .removeHead() method that has no parameters. Inside it, create a const variable named removedHead and set it equal to the list’s head.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Check if removedHead has value. If not, that means there’s nothing to remove, so return to end the method.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside of your if, set the list’s head to removedHead‘s next node.

If the head has value, set the head’s previous node to null, since the head of the list shouldn’t have a previous node.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

If removedHead is equal to the list’s tail, call the .removeTail() method (we will create this in the next exercise).

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Finally, return removedHead‘s data.

Checkpoint

Answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  // Create your .removeHead() method below:

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

Remove Tail

In addition to being able to add to the head of the list, a doubly linked list’s tail property allows us to add to the tail just as easily. In fact, like the .addToHead() and .addToTail() methods, the .removeTail() method will closely mirror the .removeHead() method:

* Start by checking if there’s a current tail to the list. If there isn’t:
  + The list is empty, so there’s nothing to return, and the method ends
* Otherwise, update the list’s tail to be the current tail’s previous node
* If there is still a tail to the list (meaning the list had more than one element when we started):
  + Set the tail’s next node to null since there should be no node after the tail of the list
* If the removed tail was also the head of the list (meaning there was only one element in the list), call .removeHead() to make the necessary changes to the head of the list
* Finally, return the old tail

**Instructions**

**1.**

Define a .removeTail() method that has no parameters. Inside it, create a const variable named removedTail and set it equal to the list’s tail.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Check if removedTail has value. If not, that means there’s nothing to remove, so return to end the method.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside of your if, set the list’s tail to removedTail‘s previous node.

If the tail has value, set the tail’s next node to null, since the tail of the list shouldn’t have a next node.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

If removedTail is equal to the list’s head, call the .removeHead() method.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Finally, return removedTail‘s data.

Checkpoint

Answer: DoublyLinkedlist.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  // Create your .removeTail() method below:

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

Remove By Data I

In addition to removing the head and the tail of the list, it would also be useful to remove a specific element that could be anywhere in the list. Imagine that you have a list of errands to run. You don’t always do your errands in order, so when you finish doing your laundry, that could be somewhere in the middle of the list. We are going to build a .removeByData() method that will allow you to cross off (remove) that errand no matter where it is in the list.

In order to do this:

* Iterate through the list to find the matching node
* If there is no matching element in the list:
  + Return null
* If there is a matching node, we will then check to see if it is the head or tail of the list:
  + If so, call the corresponding .removeHead() or .removeTail() method
* If not, that means the node was somewhere in the middle of the list. In that case:
  + Remove it by resetting the pointers of its previous and next nodes
* Finally, return the node

**Instructions**

**1.**

Define a .removeByData() method that takes data as a parameter. Inside it, create a let variable named nodeToRemove. We don’t know what it is yet, so don’t give it any value.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Create a let variable named currentNode and set it equal to the list’s head. Then create a while loop that runs while currentNode isn’t null. Inside the loop, update currentNode to be its next node. This is how we will iterate through the list as we look for the matching node.

(If you accidentally create an infinite loop and your code won’t stop running, you can reload the page to stop it.)

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Inside the while loop, but before you updated currentNode to be its next node, create an if statement that checks if currentNode‘s data matches data. If it does, that means we have found the matching node. Inside the if:

* Set nodeToRemove to currentNode
* break to leave the while loop, since we don’t need to keep looking through the list

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Outside your while loop, check if nodeToRemove has any value. If it doesn’t, that means there was no matching node in the list, so return null.

Checkpoint 5 Passed

answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  // Create your .removeByData() method below:

  removeByData(data) {

    let nodeToRemove;

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        nodeToRemove = currentNode;

        break;

      }

      currentNode = currentNode.getNextNode();

    }

    if (!nodeToRemove) {

      return null;

    }

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

Remove By Data II

Now that we’ve found the node that we want to remove from the list (or returned null if it didn’t exist), it’s time to actually remove the node. This means resetting the pointers around the node. There are three cases here:

* The node was the head of the list, in which case we can just call .removeHead()
* The node was the tail of the list, in which case we can just call .removeTail()
* The node was somewhere in the middle of the list, in which case we will need to manually change the pointers for its previous and next nodes

Let’s get started!

### Instructions

**1.**

Still in your .removeByData() method, check if nodeToRemove is the list’s head. If so, call .removeHead().

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Else, if nodeToRemove is the list’s tail, call .removeTail().

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Else, we know that the node is somewhere in the middle of the list. To remove it, we will need to reset the pointers for the nodes around it. In an else block, create:

* A const variable named nextNode that is equal to nodeToRemove‘s next node
* A const variable named previousNode that is equal to nodeToRemove‘s previous node

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Now that we have our nodes, we can remove the pointers to and from nodeToRemove and have nextNode and previousNode point to each other. Still in the else block:

* Set nextNode‘s previous node to previousNode
* Set previousNode‘s next node to nextNode

Finally, outside of the else block, return nodeToRemove.

Answer: DoubllyLInkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  removeByData(data) {

    let nodeToRemove;

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        nodeToRemove = currentNode;

        break;

      }

      currentNode = currentNode.getNextNode();

    }

    if (!nodeToRemove) {

      return null;

    }

    // Continue your .removeByData() method below:

    if (nodeToRemove === this.head) {

      this.removeHead();

    } else if (nodeToRemove === this.tail) {

      this.removeTail();

    } else {

      const nextNode = nodeToRemove.getNextNode();

      const previousNode = nodeToRemove.getPreviousNode();

      nextNode.setPreviousNode(previousNode);

      previousNode.setNextNode(nextNode);

    }

    return nodeToRemove;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

Using the Doubly Linked List

You finished your DoublyLinkedList class! Now we’re going to use that class to model a subway line. A doubly linked list is a great data structure to use to model a subway line, as both have a first and last element, and are comprised of nodes (or stops) with links to the elements before and after them.

We will add to and remove stops from our subway line, and print it out to see what we’ve done. The .printList() method is the same as the one from the LinkedList class and has been provided.

**Instructions**

**1.**

We’re going to model a (fictional) subway line that will travel from Central Park to the Brooklyn Bridge. In **subway.js** create a new DoublyLinkedList named subway.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

The subway starts at Central Park and winds its way downtown. In the following order:

* add 'TimesSquare' to the head of the list
* add 'GrandCentral' to the head of the list
* add 'CentralPark' to the head of the list

Then print the list.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

The subway continues from Times Square down to the Brooklyn Bridge. In the following order:

* Add 'PennStation' to the tail of the list
* Add 'WallStreet' to the tail of the list
* Add 'BrooklynBridge' to the tail of the list

Then print the list again.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Oh no! There’s construction happening on the subway line: the Central Park and Brooklyn Bridge stops will temporarily be closed. Remove them from your list without iterating through the entire list.

Then print the list again.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

It turns out that the Times Square station is under construction as well. Remove it from the list, and then print the list for the last time.

Checkpoint 6 Passed

answer: subway.js

const DoublyLinkedList = require('./DoublyLinkedList.js');

const subway = new DoublyLinkedList();

subway.addToHead('TimesSquare');

subway.addToHead('GrandCentral');

subway.addToHead('CentralPark');

subway.printList();

subway.addToTail('PennStation');

subway.addToTail('WallStreet');

subway.addToTail('BrooklynBridge');

subway.printList();

subway.removeHead();

subway.removeTail();

subway.printList();

subway.removeByData('TimesSquare');

subway.printList();

DoublyLnkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  removeByData(data) {

    let nodeToRemove;

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        nodeToRemove = currentNode;

        break;

      }

      currentNode = currentNode.getNextNode();

    }

    if (!nodeToRemove) {

      return null;

    }

    if (nodeToRemove === this.head) {

      this.removeHead();

    } else if (nodeToRemove === this.tail) {

      this.removeTail();

    } else {

      const nextNode = nodeToRemove.getNextNode();

      const previousNode = nodeToRemove.getPreviousNode();

      nextNode.setPreviousNode(previousNode);

      previousNode.setNextNode(nextNode);

    }

    return nodeToRemove;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

const subway = new DoublyLinkedList()

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

We did this by:

* Using our Node class to hold the data and links between nodes
* Implementing a DoublyLinkedList class to handle external operations on the list, like adding and removing nodes
* Creating an instance of our list, and using the .printList() method to track the changes we made

I**DOUBLY LINKED LISTS: CONCEPTUAL**

**Doubly Linked Lists Introduction**

Common operations on a doubly linked list may include:

* adding nodes to both ends of the list
* removing nodes from both ends of the list
* finding, and removing, a node from anywhere in the list
* traversing (or traveling through) the list

## Image

**Adding to the List**

In a doubly linked list, adding to the list is a little complicated as we have to keep track of and set the node’s previous pointer as well as update the tail of the list if necessary.

#### Adding to the Head

When adding to the head of the doubly linked list, we first need to check if there is a current head to the list. If there isn’t, then the list is empty, and we can simply make our new node both the head and tail of the list and set both pointers to null. If the list is not empty, then we will:

* Set the current head’s previous pointer to our new head
* Set the new head’s next pointer to the current head
* Set the new head’s previous pointer to null

#### Adding to the Tail

Similarly, there are two cases when adding a node to the tail of a doubly linked list. If the list is empty, then we make the new node the head and tail of the list and set the pointers to null. If the list is not empty, then we:

* Set the current tail’s next pointer to the new tail
* Set the new tail’s previous pointer to the current tail
* Set the new tail’s next pointer to null

**Removing from the Head and Tail**

* Due to the extra pointer and tail property, removing the head from a doubly linked list is slightly more complicated than removing the head from a singly linked list. However, the previous pointer and tail property make it much simpler to remove the tail of the list, as we don’t have to traverse the entire list to be able to do it.
* **Removing the Head**
* Removing the head involves updating the pointer at the beginning of the list. We will set the previous pointer of the new head (the element directly after the current head) to null, and update the head property of the list. If the head was also the tail, the tail removal process will occur as well.
* **Removing the Tail**
* Similarly, removing the tail involves updating the pointer at the end of the list. We will set the next pointer of the new tail (the element directly before the tail) to null, and update the tail property of the list. If the tail was also the head, the head removal process will occur as well.

**Removing from the Middle of the List**

It is also possible to remove a node from the middle of the list. Since that node is neither the head nor the tail of the list, there are two pointers that must be updated:

* We must set the removed node’s preceding node’s next pointer to its following node
* We must set the removed node’s following node’s previous pointer to its preceding node

There is no need to change the pointers of the removed node, as updating the pointers of its neighboring nodes will remove it from the list. If no nodes in the list are pointing to it, the node is orphaned.

**Doubly Linked Lists Review**

Let’s take a minute to review what we’ve covered about doubly linked lists in this lesson. Doubly Linked Lists:

* Are comprised of nodes that contain links to the next and previous nodes
* Are bidirectional, meaning it can be traversed in both directions
* Have a pointer to a single head node, which serves as the first node in the list
* Have a pointer to a single tail node, which serves as the last node in the list
* Require the pointers at the head of the list to be updated after addition to or removal of the head
* Require the pointers at the tail of the list to be updated after addition to or removed of the tail
* Require the pointers of the surrounding nodes to be updated after removal from the middle of the list

Your browser history is another example of a doubly linked list. When you open your browser, the page that you land on is the head of your list. As you click on things and navigate to new pages, you are moving forward and adding to the tail of your list. If you ever want to go back to something you’ve already visited, you can use the “back” button to move backward through your list. Can you think of another computer use case for a doubly linked list?

**DOUBLY LINKED LISTS: JAVASCRIPT**

**Node Class and Constructor**

Let’s implement a doubly linked list in JavaScript. A doubly linked list is a sequential chain of nodes, just like a linked list. The nodes we used for our linked lists contained two elements:

* data
* a link to the next node

The difference between a doubly linked list and a linked list is that in a doubly linked list, there are pointers to the previous node as well as the next node. This means that the doubly linked list data structure has a tail property in addition to the head property that’s present in the linked list data structure.

Depending on the end-use of the doubly linked list, there are a variety of methods that we can define.

For our use, we want to be able to:

* add a new node to the beginning (head) of the list
* add a new node to the end (tail) of the list
* remove a node from the beginning (head) of the list
* remove a node from the end (tail) of the list
* find and remove a specific node by its data
* print out the nodes in the list in order from head to tail

We will reuse the .printList() method from our LinkedList class, but the rest will either be edited or new.

To start, we are going to look at the updated Node class and create the constructor.

Ready? Let’s get started!

### Instructions

**1.**

We are going to use a provided Node class, which you can find in **Node.js**. We’ve added a previous property to the class, as well as .setPreviousNode() and .getPreviousNode() methods. Take a look at it to see the changes.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In your DoublyLinkedList class, create a constructor that has no parameters.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Inside your DoublyLinkedList constructor:

* Set the list’s head to null
* Set the list’s tail to null

Checkpoint 4 Passed

ANSWER:DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  // Create your constructor below:

  constructor() {

    this.head = null;

    this.tail = null;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

**Add To Head**

In a linked list, we add to the head of the list by checking to see if there is already a head. We then either set the new node as the head (if there was no head) or update the head property, and link the past head to the new head.

Since a doubly linked list has an additional tail property and is built with nodes that each have two pointers, there are a few more steps:

* Start by checking to see if there is a current head to the list
* If there is (meaning the list is not empty), then we want to reset the pointers at the head of the list:
  + Set the current head’s previous node to the new head
  + Set the new head’s next node to the current head
* Update the head property to be the new head
* Finally, if there isn’t a current tail to the list (meaning the list was empty):
  + Update the tail property to be the new head since that node will be both the head and tail

### Instructions

**1.**

Define an .addToHead() method that takes one parameter called data. Inside, create:

* A Node const variable named newHead that takes data as an argument
* A const variable named currentHead that’s set to the list’s head

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

If there is a current head to the list:

* Set currentHead‘s previous node to newHead
* Set newHead‘s next node to currentHead

Remember to use the Node class’s .setNextNode() and .setPreviousNode() methods.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside of the if, set the list’s head to the new head.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Lastly, if the list doesn’t have a tail, set the list’s tail to the new head.

Checkpoint 5 Passed

Answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  // Create your .addToHead() method below:

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

**Add To Tail**

Since doubly linked lists have a tail property, we don’t have to iterate through the entire list to add to the tail like we did with a singly linked list. The new method will mirror what we did in our .addToHead() method:

* Start by checking to see if there is a current tail to the list
* If there is (meaning the list is not empty), then we want to reset the pointers at the tail of the list:
  + Set the current tail’s next node to the new tail
  + Set the new tail’s previous node to the current tail
* Update the tail property to be the new tail
* Finally, if there isn’t a current head to the list (meaning the list was empty):
  + Update the head property to be the new tail since that node will be both the head and tail

### Instructions

**1.**

Define an .addToTail() method that takes one parameter called data. Inside, create:

* A Node const variable named newTail that takes data as an argument
* A const variable named currentTail, and set it equal to the list’s tail

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

If there is a current tail to the list:

* Set the current tail’s next node to newTail
* Set newTail‘s previous node to the current tail

Remember to use the Node class’s .setNextNode() and .setPreviousNode() methods.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside your if, set the list’s tail to the new tail.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Lastly, if the list doesn’t have a head, set the list’s head to the new tail.

Checkpoint

### Answer:DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  // Create your .addToTail() method below:

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    this.head = newTail;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

**Remove Head**

Due to the added tail property, removing the head of the list in a doubly linked list is a little more complicated than doing so in a singly linked list:

* Start by checking if there’s a current head to the list. If there isn’t:
  + The list is empty, so there’s nothing to return, and the method ends
* Otherwise, update the list’s head to be the current head’s next node
* If there is still a head to the list (meaning the list had more than one element when we started):
  + Set the head’s previous node to null since there should be no node before the head of the list
* If the removed head was also the tail of the list (meaning there was only one element in the list), call .removeTail() to make the necessary changes to the tail of the list. (We will create this method in the next exercise!)
* Finally, return the old head

### Instructions

**1.**

Define a .removeHead() method that has no parameters. Inside it, create a const variable named removedHead and set it equal to the list’s head.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Check if removedHead has value. If not, that means there’s nothing to remove, so return to end the method.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside of your if, set the list’s head to removedHead‘s next node.

If the head has value, set the head’s previous node to null, since the head of the list shouldn’t have a previous node.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

If removedHead is equal to the list’s tail, call the .removeTail() method (we will create this in the next exercise).

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Finally, return removedHead‘s data.

Checkpoint 6 Passed

### Answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  // Create your .removeHead() method below:

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

**Remove Tail**

In addition to being able to add to the head of the list, a doubly linked list’s tail property allows us to add to the tail just as easily. In fact, like the .addToHead() and .addToTail() methods, the .removeTail() method will closely mirror the .removeHead() method:

* Start by checking if there’s a current tail to the list. If there isn’t:
  + The list is empty, so there’s nothing to return, and the method ends
* Otherwise, update the list’s tail to be the current tail’s previous node
* If there is still a tail to the list (meaning the list had more than one element when we started):
  + Set the tail’s next node to null since there should be no node after the tail of the list
* If the removed tail was also the head of the list (meaning there was only one element in the list), call .removeHead() to make the necessary changes to the head of the list
* Finally, return the old tail

### Instructions

**1.**

Define a .removeTail() method that has no parameters. Inside it, create a const variable named removedTail and set it equal to the list’s tail.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Check if removedTail has value. If not, that means there’s nothing to remove, so return to end the method.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Outside of your if, set the list’s tail to removedTail‘s previous node.

If the tail has value, set the tail’s next node to null, since the tail of the list shouldn’t have a next node.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

If removedTail is equal to the list’s head, call the .removeHead() method.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Finally, return removedTail‘s data.

Checkpoint 6 Passed

### answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  // Create your .removeTail() method below:

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

**Remove By Data I**

In addition to removing the head and the tail of the list, it would also be useful to remove a specific element that could be anywhere in the list. Imagine that you have a list of errands to run. You don’t always do your errands in order, so when you finish doing your laundry, that could be somewhere in the middle of the list. We are going to build a .removeByData() method that will allow you to cross off (remove) that errand no matter where it is in the list.

In order to do this:

* Iterate through the list to find the matching node
* If there is no matching element in the list:
  + Return null
* If there is a matching node, we will then check to see if it is the head or tail of the list:
  + If so, call the corresponding .removeHead() or .removeTail() method
* If not, that means the node was somewhere in the middle of the list. In that case:
  + Remove it by resetting the pointers of its previous and next nodes
* Finally, return the node

### Instructions

**1.**

Define a .removeByData() method that takes data as a parameter. Inside it, create a let variable named nodeToRemove. We don’t know what it is yet, so don’t give it any value.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Create a let variable named currentNode and set it equal to the list’s head. Then create a while loop that runs while currentNode isn’t null. Inside the loop, update currentNode to be its next node. This is how we will iterate through the list as we look for the matching node.

(If you accidentally create an infinite loop and your code won’t stop running, you can reload the page to stop it.)

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Inside the while loop, but before you updated currentNode to be its next node, create an if statement that checks if currentNode‘s data matches data. If it does, that means we have found the matching node. Inside the if:

* Set nodeToRemove to currentNode
* break to leave the while loop, since we don’t need to keep looking through the list

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Outside your while loop, check if nodeToRemove has any value. If it doesn’t, that means there was no matching node in the list, so return null.

Checkpoint 5 Passed

### answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  // Create your .removeByData() method below:

  removeByData(data) {

    let nodeToRemove;

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        nodeToRemove = currentNode;

        break;

      }

      currentNode = currentNode.getNextNode();

    }

    if (!nodeToRemove) {

      return null;

    }

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

**Remove By Data II**

Now that we’ve found the node that we want to remove from the list (or returned null if it didn’t exist), it’s time to actually remove the node. This means resetting the pointers around the node. There are three cases here:

* The node was the head of the list, in which case we can just call .removeHead()
* The node was the tail of the list, in which case we can just call .removeTail()
* The node was somewhere in the middle of the list, in which case we will need to manually change the pointers for its previous and next nodes

Let’s get started!

### Instructions

**1.**

Still in your .removeByData() method, check if nodeToRemove is the list’s head. If so, call .removeHead().

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Else, if nodeToRemove is the list’s tail, call .removeTail().

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Else, we know that the node is somewhere in the middle of the list. To remove it, we will need to reset the pointers for the nodes around it. In an else block, create:

* A const variable named nextNode that is equal to nodeToRemove‘s next node
* A const variable named previousNode that is equal to nodeToRemove‘s previous node

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Now that we have our nodes, we can remove the pointers to and from nodeToRemove and have nextNode and previousNode point to each other. Still in the else block:

* Set nextNode‘s previous node to previousNode
* Set previousNode‘s next node to nextNode

Finally, outside of the else block, return nodeToRemove.

Checkpoint 5 Passed

### Answer: DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  removeByData(data) {

    let nodeToRemove;

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        nodeToRemove = currentNode;

        break;

      }

      currentNode = currentNode.getNextNode();

    }

    if (!nodeToRemove) {

      return null;

    }

    // Continue your .removeByData() method below:

    if (nodeToRemove === this.head) {

      this.removeHead();

    } else if (nodeToRemove === this.tail) {

      this.removeTail();

    } else {

      const nextNode = nodeToRemove.getNextNode();

      const previousNode = nodeToRemove.getPreviousNode();

      nextNode.setPreviousNode(previousNode);

      previousNode.setNextNode(nextNode);

    }

    return nodeToRemove;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

module.exports = DoublyLinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

**Using the Doubly Linked List**

You finished your DoublyLinkedList class! Now we’re going to use that class to model a subway line. A doubly linked list is a great data structure to use to model a subway line, as both have a first and last element, and are comprised of nodes (or stops) with links to the elements before and after them.

We will add to and remove stops from our subway line, and print it out to see what we’ve done. The .printList() method is the same as the one from the LinkedList class and has been provided.

### Instructions

**1.**

We’re going to model a (fictional) subway line that will travel from Central Park to the Brooklyn Bridge. In **subway.js** create a new DoublyLinkedList named subway.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

The subway starts at Central Park and winds its way downtown. In the following order:

* add 'TimesSquare' to the head of the list
* add 'GrandCentral' to the head of the list
* add 'CentralPark' to the head of the list

Then print the list.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

The subway continues from Times Square down to the Brooklyn Bridge. In the following order:

* Add 'PennStation' to the tail of the list
* Add 'WallStreet' to the tail of the list
* Add 'BrooklynBridge' to the tail of the list

Then print the list again.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Oh no! There’s construction happening on the subway line: the Central Park and Brooklyn Bridge stops will temporarily be closed. Remove them from your list without iterating through the entire list.

Then print the list again.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

It turns out that the Times Square station is under construction as well. Remove it from the list, and then print the list for the last time.

Checkpoint 6 Passed

### answer; subway.js

const DoublyLinkedList = require('./DoublyLinkedList.js');

const subway = new DoublyLinkedList();

subway.addToHead('TimesSquare');

subway.addToHead('GrandCentral');

subway.addToHead('CentralPark');

subway.printList();

subway.addToTail('PennStation');

subway.addToTail('WallStreet');

subway.addToTail('BrooklynBridge');

subway.printList();

subway.removeHead();

subway.removeTail();

subway.printList();

subway.removeByData('TimesSquare');

subway.printList();

### DoublyLinkedList.js

const Node = require('./Node');

class DoublyLinkedList {

  constructor() {

    this.head = null;

    this.tail = null;

  }

  addToHead(data) {

    const newHead = new Node(data);

    const currentHead = this.head;

    if (currentHead) {

      currentHead.setPreviousNode(newHead);

      newHead.setNextNode(currentHead);

    }

    this.head = newHead;

    if (!this.tail) {

      this.tail = newHead;

    }

  }

  addToTail(data) {

    const newTail = new Node(data);

    const currentTail = this.tail;

    if (currentTail) {

      currentTail.setNextNode(newTail);

      newTail.setPreviousNode(currentTail);

    }

    this.tail = newTail;

    if (!this.head) {

      this.head = newTail;

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    if (this.head) {

      this.head.setPreviousNode(null);

    }

    if (removedHead === this.tail) {

      this.removeTail();

    }

    return removedHead.data;

  }

  removeTail() {

    const removedTail = this.tail;

    if (!removedTail) {

      return;

    }

    this.tail = removedTail.getPreviousNode();

    if (this.tail) {

      this.tail.setNextNode(null);

    }

    if (removedTail === this.head) {

      this.removeHead();

    }

    return removedTail.data;

  }

  removeByData(data) {

    let nodeToRemove;

    let currentNode = this.head;

    while (currentNode !== null) {

      if (currentNode.data === data) {

        nodeToRemove = currentNode;

        break;

      }

      currentNode = currentNode.getNextNode();

    }

    if (!nodeToRemove) {

      return null;

    }

    if (nodeToRemove === this.head) {

      this.removeHead();

    } else if (nodeToRemove === this.tail) {

      this.removeTail();

    } else {

      const nextNode = nodeToRemove.getNextNode();

      const previousNode = nodeToRemove.getPreviousNode();

      nextNode.setPreviousNode(previousNode);

      previousNode.setNextNode(nextNode);

    }

    return nodeToRemove;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.getNextNode();

    }

    output += '<tail>';

    console.log(output);

  }

}

const subway = new DoublyLinkedList()

module.exports = DoublyLinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

    this.previous = null;

  }

  setNextNode(node) {

    if (node instanceof Node || node === null) {

      this.next = node;

    } else {

      throw new Error('Next node must be a member of the Node class')

    }

  }

  setPreviousNode(node) {

    if (node instanceof Node || node === null) {

      this.previous = node;

    } else {

      throw new Error('Previous node must be a member of the Node class')

    }

  }

  getNextNode() {

    return this.next;

  }

  getPreviousNode() {

    return this.previous;

  }

}

module.exports = Node;

* Using our Node class to hold the data and links between nodes
* Implementing a DoublyLinkedList class to handle external operations on the list, like adding and removing nodes
* Creating an instance of our list, and using the .printList() method to track the changes we made

# Queues

A queue is a linear collection of nodes that exclusively adds (enqueues) nodes to the tail, and removes (dequeues) nodes from the head of the queue. They can be implemented using different underlying data structures, but one of the more common methods is to use a singly linked list, which is what you will be using for your JavaScript **Queue** class. Think of the queue data structure as an actual queue, or line, in a grocery store. The person at the front gets to leave the line first, and every person who joins the line has to join in the back.

**QUEUES: CONCEPTUAL**

**Queues Introduction**

A queue is a data structure which contains an ordered set of data.

Queues provide three methods for interaction:

* Enqueue - adds data to the “back” or end of the queue
* Dequeue - provides and removes data from the “front” or beginning of the queue
* Peek - reveals data from the “front” of the queue without removing it

This data structure mimics a physical queue of objects like a line of people buying movie tickets. Each person has a name (the data). The first person to *enqueue*, or get into line, is both at the front and back of the line. As each new person enqueues, they become the new back of the line.

When the cashier serves someone, they begin at the front of the line (or people would get very mad!). Each person served is *dequeued* from the front of the line, they purchase a ticket and leave.

If they just want to know who is next in line, they can *peek* and get their name **without removing them from the queue.**

The first person in the queue is the first to be served. Queues are a First In, First Out or FIFO structure.

**Queues Implementation**

Queues can be implemented using a linked list as the underlying data structure. The front of the queue is equivalent to the head node of a linked list and the back of the queue is equivalent to the tail node.

Since operations are only allowed affecting the front or back of the queue, any traversal or modification to other nodes within the linked list is disallowed. Since both ends of the queue must be accessible, a reference to both the head node and the tail node must be maintained.

One last constraint that may be placed on a queue is its length. If a queue has a limit on the amount of data that can be placed into it, it is considered a *bounded queue*.

Similar to stacks, attempting to enqueue data onto an already full queue will result in a *queue overflow*. If you attempt to dequeue data from an empty queue, it will result in a *queue underflow*.

**Queues Review**

Let’s take a minute to review what we’ve covered about queues in this lesson.

Queues:

* Contain data nodes
* Support three main operations:
  + Enqueue adds data to the back of the queue
  + Dequeue removes and provides data from the front of the queue
  + Peek provides data on the front of the queue
* Can be implemented using a linked list or array
* Bounded queues have a limited size.
* Enqueueing onto a full queue causes a queue overflow
* Queues process data First In, First Out (FIFO)

**LEARN QUEUES: JAVASCRIPT**

**Introduction: Queues in JavaScript**

You can visualize it as a checkout line at a supermarket:

* The customer at the front of the line (equivalent to the head in a queue) is the first customer to pay for their groceries.
* Any new customer must go to the back of the line (the tail of the queue) and wait until everyone in front of them has paid for their groceries, (no line cutters allowed in this supermarket!)
* The supermarket cashier only needs to check out the customer at the front of the line

Real-life computer science applications of queues are all around us: search algorithms like BFS (breadth-first search), job schedulers that run tasks on our computers, and keyboard processing that interprets our keystrokes are all queue based.

We’ll also set up a few helper methods that will help us keep track of the queue size in order to prevent queue overflow and underflow.

To do this, we’ll make use of some data structures you’ve already seen: nodes and linked lists. Feel free to take a look at the **Node.js** and **LinkedList.js** files in the code editor to review how these data structures are implemented and see what methods are available to you to use in this lesson’s exercises.

### Instructions

**1.**

Within **Queue.js** and inside of the Queue class, we’ve given you a .constructor() method. Since all new queues are empty, they have 0 nodes.

Inside of the .constructor(), create a property that tracks the number of elements in a queue and call it size.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Check your work. Uncomment the code at the bottom of the file, run it, and read the message logged to the terminal.

Checkpoint

Answer: Queqe.js

const LinkedList = require('./LinkedList');

class Queue {

  constructor() {

    this.queue = new LinkedList();

    this.size = 0;

  }

}

module.exports = Queue;

const restaurantOrders = new Queue();

console.log(`restaurantOrders has ${restaurantOrders.size} nodes.`)

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Queue Methods: Enqueue**

*Enqueue* is a fancy way of saying “add to a queue,” and that is exactly what we’re doing with the .enqueue() method.

When adding a node to a queue, the new node is always added to the end of the queue. If the queue is empty, the node we’re adding becomes both the head and tail of the queue. If the queue has at least one other node, the added node only becomes the new tail.

Let’s put this into action by building out an .enqueue() method for Queue.

**Instructions**

**1.**

Inside the Queue class you built, define an instance method .enqueue() that takes a value, data, as a parameter.

Checkpoint 2 Passed

**2.**

Add a new node to the queue. Do this by calling the method that adds a new value to the end of the underlying linked list.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

The size property tracks the number of nodes in a queue. It should increase as nodes are added and decrease as nodes are removed.

Add code to increment the queue’s size by 1 every time a node is added to the queue.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

If the enqueue was successful, add code in the .enqueue() method that logs a message telling us what was added and the new size of the queue. If we ran the following code:

const groceries = new Queue();  
groceries.enqueue('eggs');

It should log a message that tells us what was added and the new size of the queue:

Added eggs! Queue size is now 1.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Check your work. Uncomment the code at the bottom of the file to run it and read the messages logged to the terminal.

Checkpoint 6 Passed

Answer: Queqe.js

const LinkedList = require('./LinkedList');

class Queue {

  constructor() {

    this.queue = new LinkedList();

    this.size = 0;

  }

  enqueue(data) {

    this.queue.addToTail(data);

    this.size++;

    console.log(`Added ${data}! Queue size is now ${this.size}.`);

  }

}

module.exports = Queue;

const restaurantOrder = new Queue();

console.log(`restaurantOrder queue has ${restaurantOrder.size} orders.\n`);

restaurantOrder.enqueue('apple pie');

restaurantOrder.enqueue('roast chicken');

restaurantOrder.enqueue('quinoa salad');

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Queue Methods: Dequeue**

We can add items to the tail of our queue, but we remove them from the head using a method known as .dequeue(), which is another way to say “remove from a queue.”

Our .dequeue() removes the current head and replaces it with the following node. The .dequeue() method should also return the value of the head node.

If the queue has one node, when we remove it, the queue will be empty. If the queue has more than one node, we just remove the head node and reset the head to the following node.

### Instructions

**1.**

Inside the Queue class you built, define a method .dequeue() that:

* Gets the value from the correct node using a LinkedList method
* Stores the value in a constant called data
* Finally, returns the value stored in data

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Since we’re removing a node from the queue, we’ll need to update the queue’s size. Decrement the queue’s size by 1.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

If the dequeue was successful, log a message that tells us what was removed and the new size of the queue.

If we ran the following code:

groceries = new Queue();  
groceries.enqueue('eggs');  
groceries.dequeue();

It should return this string that tells us what was added and the new size of the queue:

Removed eggs! Queue size is 0.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Uncomment the code at the bottom of the text editor, run it, and read the messages logged in the terminal.

Checkpoint 5 Passed

answer: Queqe.js

const LinkedList = require('./LinkedList');

class Queue {

  constructor() {

    this.queue = new LinkedList();

    this.size = 0;

  }

  enqueue(data) {

    this.queue.addToTail(data);

    this.size++;

    console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);

  }

  dequeue() {

    const data = this.queue.removeHead();

    this.size--;

    console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);

    return data;

  }

}

module.exports = Queue;

const restaurantOrder = new Queue();

restaurantOrder.enqueue('apple pie');

restaurantOrder.enqueue('roast chicken');

restaurantOrder.enqueue('quinoa salad');

console.log('\nFood preparing...\n')

restaurantOrder.dequeue();

restaurantOrder.dequeue();

restaurantOrder.dequeue();

console.log('All orders ready!')

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Bounded Queues**

Some queues require limits on the number of nodes they can have, while other queues don’t. Queues that restrict the number of elements they can store are called bounded queues.

Let’s make our queue a bounded queue. To account for this, we will need to make some modifications to our Queue class so that we can keep track of and limit size where needed.

We’ll be adding a new property to help us out here:

* .maxSize, a property that bounded queues can utilize to limit the total node count

In addition, we will add two new methods:

* .hasRoom() returns true if the queue has space to add another node
* .isEmpty() returns true if the size of a queue is 0

### Instructions

**1.**

Currently, our Queue class only creates unbounded queues or queues without size limits. We will add code to the Queue .constructor() that lets us specify the maximum size of a queue when it’s first created but will otherwise default to Infinity, a type of numeric value in JavaScript.

Because Infinity doesn’t have the same behavior as and is larger than other numbers, using it is useful as a default value where it acts as a threshold or ceiling we don’t want to pass.

Add a new parameter maxSize to your .constructor() method that has a default value of Infinity. This ensures that if we don’t specify a maximum size we’ll create an unbounded queue.

Then store the maxSize parameter in a property also called .maxSize.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Below .constructor(), define a helper method .hasRoom() that checks if the current size of the queue is less than the maximum size. It should return true if the size of the queue is less than the maximum size and false if it is equal to or greater than the maximum size.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Define another method .isEmpty() for Queue. This helper method should return true if the queue has no elements stored, (if the size of the queue is 0) or false if the queue has 1 or more elements.

Checkpoint 4 Passed

Answer: Queqe.js

const LinkedList = require('./LinkedList');

class Queue {

  constructor(maxSize = Infinity) {

    this.queue = new LinkedList();

    this.maxSize = maxSize;

    this.size = 0;

  }

  isEmpty() {

    return this.size === 0;

  }

  hasRoom() {

    return this.size < this.maxSize;

  }

  enqueue(data) {

    this.queue.addToTail(data);

    this.size++;

    console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);

  }

  dequeue() {

    const data = this.queue.removeHead();

    this.size--;

    console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);

    return data;

  }

}

module.exports = Queue;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Avoiding Underflow**

There are two conditions when enqueuing and dequeuing that we should be aware of and avoid: *underflow* and *overflow*.

Underflow occurs when we try to remove elements from an already empty queue – we cannot remove a node if it doesn’t exist. Underflow affects queues whether they are bounded or unbounded.

Let’s add code that will check for underflow when we attempt to dequeue.

**Instructions**

**1.**

Let’s add logic inside of .dequeue() that will help us to avoid underflow.

Inside of the .dequeue() method, add code that checks if the queue is empty.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

If the queue isn’t empty, the head node should be removed and the size of the queue should decrease. We will also log a message if the dequeue was successful.

Move the code inside .dequeue() that already does this into the if clause you added.

Checkpoint 3 Passed

**3.**

If the queue is empty, we should let the user know we cannot dequeue a node with an error message.

Add an else branch with code that throws an error with the message Queue is empty!.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Watch your code in action. Uncomment the bounded queue at the bottom of the file.

Dequeue as many nodes as you can from boundedQueue, reading any messages that appear in the terminal including errors.

Checkpoint 5 Passed

### Answer: Queqe.js

const LinkedList = require("./LinkedList");

class Queue {

  constructor(maxSize = Infinity) {

    this.queue = new LinkedList();

    this.maxSize = maxSize;

    this.size = 0;

  }

  isEmpty() {

    return this.size === 0;

  }

  hasRoom() {

    return this.size < this.maxSize;

  }

  enqueue(data) {

    this.queue.addToTail(data);

    this.size++;

    console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);

  }

  dequeue() {

    if (!this.isEmpty()) {

    const data = this.queue.removeHead();

    this.size--;

    console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);

    return data;

  } else {

    throw new Error("Queue is empty!");

  }

  }

}

module.exports = Queue;

const boundedQueue = new Queue(3);

 boundedQueue.enqueue(1);

 boundedQueue.enqueue(2);

 boundedQueue.enqueue(3);

### LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

}

module.exports = LinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Avoiding Overflow**

Overflow occurs when we add an element to a queue that does not have room for a new node.

This condition affects bounded queues because they have fixed sizes they cannot exceed. For unbounded queues, though they don’t have a size restriction, at some point the size of the queue will exceed the available memory we can use to store this queue.

We’ll be adding code to our Queue class to check for overflow whenever we try to add a node to a queue.

**Instructions**

**1.**

Let’s add logic inside of .enqueue() to help us avoid overflow.

Inside of the .enqueue() method, add code that checks if there is room in the queue to add a new node.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

If there is room in the queue, a new node should be added to the tail end of the queue and increasing its size. We’ll also log a message if successful.

Move the code in .enqueue() that does this into the if clause.

Checkpoint 3 Passed

**3.**

If there isn’t room in the queue to add a node, we should let the user know with an error message.

Add an else branch and add code that throws a new Error with the message Queue is full!.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Watch your code in action. Uncomment the bounded queue at the bottom of the file.

Enqueue as many nodes as you can to boundedQueue, reading any messages that appear in the terminal including errors.

Dequeue as many nodes as you can from boundedQueue, reading any messages that appear in the terminal including errors.

Checkpoint 5 Passed

### Answer: Queqe.js

const LinkedList = require('./LinkedList');

class Queue {

  constructor(maxSize = Infinity) {

    this.queue = new LinkedList();

    this.maxSize = maxSize;

    this.size = 0;

  }

  isEmpty() {

    return this.size === 0;

  }

  hasRoom() {

    return this.size < this.maxSize;

  }

  enqueue(data) {

    if (this.hasRoom()) {

    this.queue.addToTail(data);

    this.size++;

    console.log(`Added ${data} to queue! Queue size is now ${this.size}.`);

  } else {

    throw new Error("Queue is full!");

  }

  }

  dequeue() {

    if (!this.isEmpty()) {

      const data = this.queue.removeHead();

      this.size--;

      console.log(`Removed ${data} from queue! Queue size is now ${this.size}.`);

      return data;

    } else {

      throw new Error("Queue is empty!");

    }

  }

}

module.exports = Queue;

 const boundedQueue = new Queue(3);

 boundedQueue.enqueue(1);

 boundedQueue.enqueue(2);

 boundedQueue.enqueue(3);

 boundedQueue.dequeue();

 boundedQueue.dequeue();

 boundedQueue.dequeue();

 boundedQueue.dequeue();

### LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(data) {

    const nextNode = new Node(data);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(data) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(data);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(data));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) {

      return;

    }

    this.head = removedHead.getNextNode();

    return removedHead.data;

  }

  printList() {

    let currentNode = this.head;

    let output = '<head> ';

    while (currentNode !== null) {

      output += currentNode.data + ' ';

      currentNode = currentNode.next;

    }

    output = output.concat("<tail>");

    console.log(output);

  }

}

module.exports = LinkedList;

### Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Review: Queues in JavaScript**

* follows FIFO protocol with .enqueue() and .dequeue() methods
* gives you the option of creating bounded queues with a .maxSize property
* prevents queue overflow and underflow by keeping track of the queue size

**STACKS: CONCEPTUAL**

**Stacks Introduction**

A stack is a data structure which contains an ordered set of data.

Stacks provide three methods for interaction:

* Push - adds data to the “top” of the stack
* Pop - returns and removes data from the “top” of the stack
* Peek - returns data from the “top” of the stack without removing it

**Stacks Implementation**

* Stacks can be implemented using a linked list as the underlying data structure because it’s more efficient than a list or array.
* Depending on the implementation, the top of the stack is equivalent to the head node of a linked list and the bottom of the stack is equivalent to the tail node.
* A constraint that may be placed on a stack is its size. This is done to limit and quantify the resources the data structure will take up when it is “full”.
* Attempting to push data onto an already full stack will result in a *stack overflow*. Similarly, if you attempt to pop data from an empty stack, it will result in a *stack underflow*.

**Stacks Review**

Let’s take a minute to review what we’ve covered about stacks in this lesson.

Stacks:

* Contain data nodes
* Support three main operations
  + Push adds data to the top of the stack
  + Pop removes and provides data from the top of the stack
  + Peek reveals data on the top of the stack
* Implementations include a linked list or array
* Can have a limited size
  + Pushing data onto a full stack results in a stack overflow
* Stacks process data Last In, First Out (LIFO)

**Introduction**

You have an understanding of how stacks work in theory, so now let’s see how they can be useful out in the wild — with Javascript!

Remember that there are three main methods that we want our stacks to have:

* Push - adds data to the top of the stack
* Pop - provides and removes data from the top of the stack
* Peek - provides data from the top of the stack without removing it

We also need to consider the stack’s size and tweak our methods a bit so that our stack does not overflow.

Let’s get started building out our Stack class.

### Instructions

**1.**

In **Stack.js**, an empty Stack class has been created for you. Inside the .constructor() method, instantiate the .stack property with a construction of a new LinkedList class that has been imported for you.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Below .constructor(), define another method .peek() that returns the data assigned to the stack’s top element. Remember that stack is a LinkedList instance that is made up of Node instances. The LinkedList class has a head property pointing to a Node instance. To access data, you first have to get to the head property.

Answer: stack.js

const LinkedList = require('./LinkedList');

class Stack {

  constructor() {

    this.stack = new LinkedList();

  }

  peek() {

    return this.stack.head.data;

  }

}

module.exports = Stack;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(value) {

    const nextNode = new Node(value);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(value) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(value);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(value));

    }

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) return;

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

}

module.exports = LinkedList;

**Push and Pop**

The stack’s .push() and .pop() methods are our tools to add and remove items from it. .pop() additionally returns the value of the item it is removing. Keep in mind that we can only make modifications to the top of the stack.

### Instructions

**1.**

Below .constructor(), define a method .push() for Stack that takes the parameter value. Inside .push(), use a LinkedList method to add a new value to the head of this.stack.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Below .push(), define a .pop() method for Stack. Inside .pop():

* figure out which of the LinkedList class methods will remove the head of the list and return its value.
* create a const variable value and set it equal to the removed value.
* return the value.

Answer: Stack.js

const LinkedList = require('./LinkedList');

class Stack {

  constructor() {

    this.stack = new LinkedList();

  }

push(value) {

  this.stack.addToHead(value);

}

pop() {

  const value = this.stack.removeHead();

  return value;

}

  peek() {

    return this.stack.head.data;

  }

}

module.exports = Stack;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(value) {

    const nextNode = new Node(value);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(value) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(value);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(value));

    }

  }

  findNodeIteratively(comparator) {

    let current = this.head;

    while (current) {

      if (comparator(current.value)) {

        return current;

      }

      current = current.getNextNode();

    }

    return null;

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) return;

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  get size() {

    let count = 0;

    let currentNode = this.head;

    while (currentNode !== null) {

      count++;

      currentNode = currentNode.next;

    }

    return count;

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Size I**

With stacks, size matters. If we’re not careful, we can accidentally over-fill them with data. Since we don’t want any stack overflow, we need to go back and make a few modifications to our methods that help us track and limit the stack size so we can keep our stacks healthy.

What do we do if someone tries to .peek() or .pop() when our stack is empty?

How do we keep someone from .push()ing to a stack that has already reached its limit?

How do we even know how large our stack has gotten?

### Instructions

**1.**

Let’s begin tracking the size of our stack. In .constructor(), we’ll add a new property size and initialize it to 0.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In the same .constructor(), let’s add:

* a parameter maxSize with a default value of Infinity so that the stack is unlimited unless we define a size for it upon instantiation later.
* a property .maxSize and initialize it to the parameter of the same name.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Let’s make sure that .peek() returns a valid value when it is not empty. In .peek(), wrap the current body with an if statement that checks if the size of the stack is greater than 0.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

If the stack is empty, we need to return null instead. In .peek() outside the if statement, add an else statement to accomplish this.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Similarly in .pop(), we want to check if the stack is empty before returning a value. Wrap the current body in an if statement that checks if the size of the stack is greater than 0.

Checkpoint 6 Passed

Stuck? Get a hint

**6.**

In .pop(), just before the return statement, reduce the size of the stack by 1.

Checkpoint 7 Passed

**7.**

If the Stack is empty, we want to let the user know with a message. Add an else statement that logs a message 'Stack is empty.'.

Answer: Stack.js

const LinkedList = require('./LinkedList');

class Stack {

  constructor(maxSize = Infinity) {

    this.stack = new LinkedList();

    this.size = 0;

    this.maxSize = maxSize;

  }

  push(value) {

    this.stack.addToHead(value);

  }

  pop() {

    if (this.size > 0) {

      const value = this.stack.removeHead();

      this.size --;

      return value;

    } else {

      console.log('Stack is empty');

    }

  }

  peek() {

    if (this.size > 0) {

      return this.stack.head.data;

    } else {

      return null;

    }

  }

}

module.exports = Stack;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(value) {

    const nextNode = new Node(value);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(value) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(value);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(value));

    }

  }

  findNodeIteratively(comparator) {

    let current = this.head;

    while (current) {

      if (comparator(current.value)) {

        return current;

      }

      current = current.getNextNode();

    }

    return null;

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) return;

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  get size() {

    let count = 0;

    let currentNode = this.head;

    while (currentNode !== null) {

      count++;

      currentNode = currentNode.next;

    }

    return count;

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Size II**

It’s time to add a couple helper methods.

Helper methods simplify the code we’ve written by abstracting and labeling chunks of code into a new function.

First, we want one that checks if our stack has room for more items. We can use this in .push() to guard against pushing items to our stack when it’s full.

Second, it’s helpful to have a method that checks if the stack is empty…

### Instructions

**1.**

Define a new method .hasRoom() in Stack. The method should return true if its current size is less than its maximum size, otherwise false.

Checkpoint 2 Passed

**2.**

Go back to your .push() method — we need to make sure we’re keeping track of our stack size when we add new items. At the end of your method body, increment this.size by 1.

Checkpoint 3 Passed

**3.**

Now add a conditional statement at the top of .push() that checks if your stack has room (using your newly created helper method).

* If there’s room, the rest of the body of the method should execute
* If there’s no room, we want to throw an Error() letting users know that the stack is already full with a message of 'Stack is full'.

Checkpoint 4 Passed

**4.**

Finally, let’s define a new method .isEmpty() in Stack.

The method should return true if the stack’s size is 0 and false otherwise.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Now we can use our .isEmpty() method in our class. Rewrite your .pop() and .peek() methods to use .isEmpty() in their conditionals.

Checkpoint 6 Passe

Answer: Stack.js

const LinkedList = require('./LinkedList');

class Stack {

  constructor(maxSize = Infinity) {

    this.stack = new LinkedList();

    this.maxSize = maxSize;

    this.size = 0;

  }

  hasRoom() {

    return (this.size < this.maxSize);

  }

  isEmpty() {

    return (this.size === 0);

  }

  push(value) {

    if (this.hasRoom()) {

      this.stack.addToHead(value);

      this.size++;

    } else {

      throw new Error('Stack is full');

    }

  }

  pop() {

    if (!this.isEmpty()) {

      const value = this.stack.removeHead();

      this.size--;

      return value;

    } else {

      throw new Error('Stack is empty!');

    }

  }

  peek() {

    if (!this.isEmpty()) {

      return this.stack.head.data;

    } else {

      return null;

    }

  }

}

module.exports = Stack;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(value) {

    const nextNode = new Node(value);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(value) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(value);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(value));

    }

  }

  findNodeIteratively(comparator) {

    let current = this.head;

    while (current) {

      if (comparator(current.value)) {

        return current;

      }

      current = current.getNextNode();

    }

    return null;

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) return;

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  get size() {

    let count = 0;

    let currentNode = this.head;

    while (currentNode !== null) {

      count++;

      currentNode = currentNode.next;

    }

    return count;

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;

**Review**

Nice work — you’ve built out a Stack class that can:

* add a new item to the top via a .push() method
* remove an item from the top and returns its value with a .pop() method
* return the value of the top item using a .peek() method
* allow a stack instance to maintain an awareness of its size to prevent stack overflow.
* **1.**
* In **main.js** file, instantiate a Stack class with a size of 6 and assign it to a variable pizzaStack using the const keyword.
* Checkpoint 2 Passed
* Stuck? Get a hint
* **2.**
* Use a for loop to push six pizzas to pizzaStack, naming each pizza 'Pizza #n' where n is from 1 to 6.
* Checkpoint 3 Passed
* Stuck? Get a hint
* **3.**
* Try adding ‘Pizza #7’ to pizzaStack to see if the program will throw any error. Add this code inside a try and catch block.
* Checkpoint 4 Passed
* Stuck? Get a hint
* **4.**
* Before we deliver our pizzas, let’s take a peek at which pizza is at the top of the stack. Log a message The first pizza to deliver is followed by the value of the top pizza.
* Checkpoint 5 Passed
* **5.**
* We are ready to deliver the pizzas off our stack. Deliver all the pizzas from the stack from top down. Which method would you use to do so?
* Checkpoint 6 Passed
* Stuck? Get a hint
* **6.**
* Is the pizza stack now empty? Try removing one more pizza off the stack. Catch any error that might be thrown. Do this in a try and catch block.
* Checkpoint

Main.js

const Stack = require('./Stack');

// 1. Define an empty pizza stack with a maxSize of 6

const pizzaStack = new Stack(6);

// 2. Add pizzas as they are ready until we fill up the stack

for (let i=1; i < 7; i++) {

  pizzaStack.push('Pizza #'+i);

}

// 3. Try pushing another pizza to check for overflow

try {

  pizzaStack.push('Pizza #7');

} catch(e) {

  console.log(e);

}

// 4. Peek at the pizza on the top of stack and log its value

console.log('The first pizza to deliver is', pizzaStack.peek());

// 5. Deliver all the pizzas from the top of the stack down

for (let i=0; i < 6; i++) {

  pizzaStack.pop();

}

// 6. Try popping another pizza to check for empty stack

try {

  pizzaStack.pop();

} catch(e) {

  console.log(e);

}

Stack.js

const LinkedList = require('./LinkedList');

class Stack {

  constructor(maxSize = Infinity) {

    this.stack = new LinkedList();

    this.maxSize = maxSize;

    this.size = 0;

  }

  // Add helper methods below this line

  hasRoom() {

    return this.size < this.maxSize;

  }

  isEmpty() {

    return this.size === 0;

  }

  push(value) {

    if (this.hasRoom()) {

      this.stack.addToHead(value);

      this.size++;

    } else {

      throw new Error('Stack is full');

    }

  }

  pop() {

    if (!this.isEmpty()) {

      const value = this.stack.removeHead();

      this.size--;

      return value;

    } else {

      throw new Error('Stack is empty');

    }

  }

  peek() {

    if (!this.isEmpty()) {

      return this.stack.head.data;

    } else {

      return null;

    }

  }

}

module.exports = Stack;

LinkedList.js

const Node = require('./Node');

class LinkedList {

  constructor() {

    this.head = null;

  }

  addToHead(value) {

    const nextNode = new Node(value);

    const currentHead = this.head;

    this.head = nextNode;

    if (currentHead) {

      this.head.setNextNode(currentHead);

    }

  }

  addToTail(value) {

    let lastNode = this.head;

    if (!lastNode) {

      this.head = new Node(value);

    } else {

      let temp = this.head;

      while (temp.getNextNode() !== null) {

        temp = temp.getNextNode();

      }

      temp.setNextNode(new Node(value));

    }

  }

  findNodeIteratively(comparator) {

    let current = this.head;

    while (current) {

      if (comparator(current.value)) {

        return current;

      }

      current = current.getNextNode();

    }

    return null;

  }

  removeHead() {

    const removedHead = this.head;

    if (!removedHead) return;

    if (removedHead.next) {

      this.head = removedHead.next;

    }

    return removedHead.data;

  }

  get size() {

    let count = 0;

    let currentNode = this.head;

    while (currentNode !== null) {

      count++;

      currentNode = currentNode.next;

    }

    return count;

  }

}

module.exports = LinkedList;

Node.js

class Node {

  constructor(data) {

    this.data = data;

    this.next = null;

  }

  setNextNode(node) {

    if (!(node instanceof Node)) {

      throw new Error('Next node must be a member of the Node class');

    }

    this.next = node;

  }

  setNext(data) {

    this.next = data;

  }

  getNextNode() {

    return this.next;

  }

}

module.exports = Node;