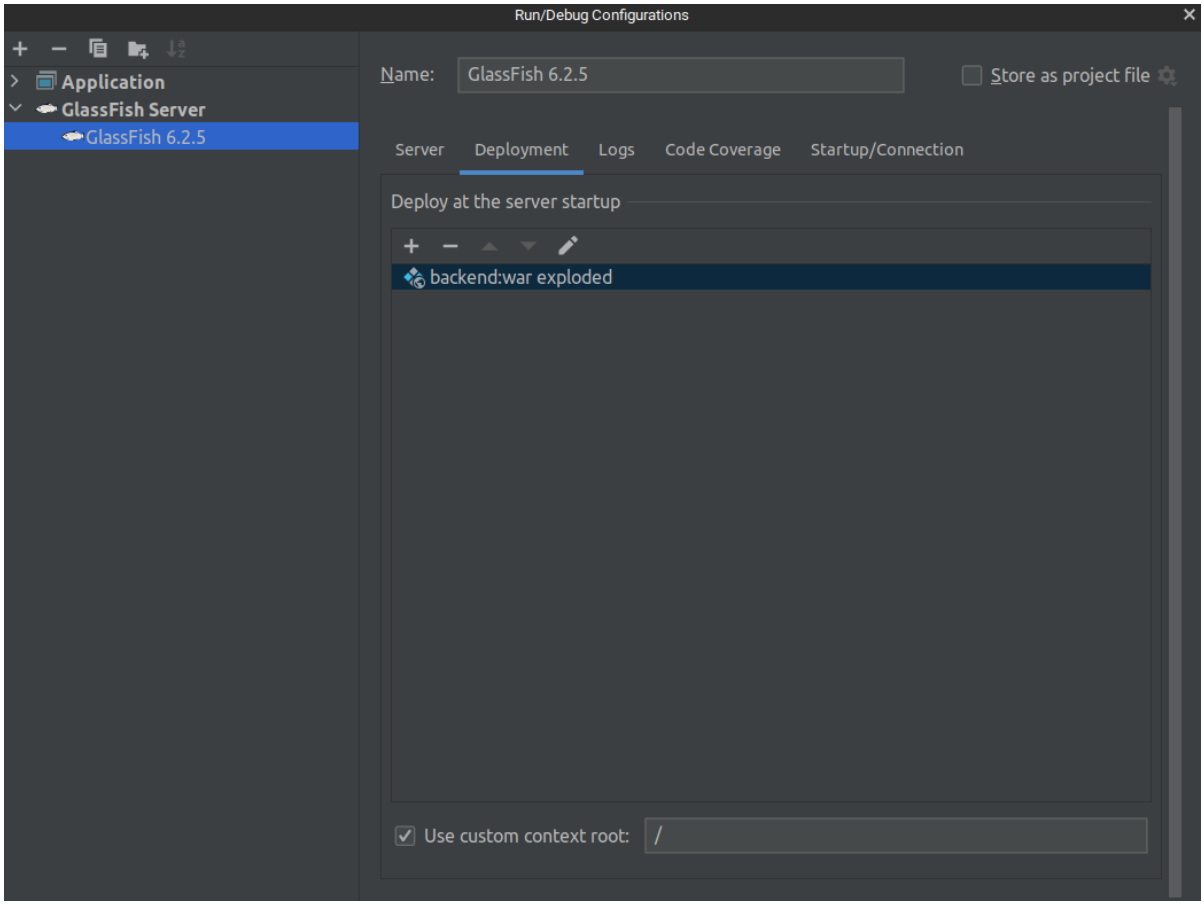
ESD Full Stack Integration

Use Case

Description of the use case that's considered with the code sections below and in the repository.
Two entities, Students & Bills
A Student has any number of Bills.
A Bill is only associated with a Student.
A Student can pay the full Bill amount for any Bill in 1 transaction, so partial Bill payments are not allowed and only one Bill can be paid at a time (which makes things easy from an implementation point of view).
A Student must first login before any bills can be paid.

Overall flow:
Login => Show Bills => Click Pay button next to any Bill to pay that Bill.

GitHub Link: https://github.com/james-jasvin/ESD-Fullstack-Integration
Note: Your GlassFish server deployment tab should look like this, note the "Use custom context root" option at the bottom.



Frontend

Installations

Install VSCode, https://code.visualstudio.com/

Install the following VSCode extensions,
ESLint (for semantic errors and syntax highlighting in JavaScript)
REST Client (might not be useful now but certainly in the future)

Install the following browser extension for analyzing and debugging components and states (again maybe not useful for now but very helpful in general),
React Developer Tools

You can now download the zip file of the GitHub repository or clone it as per your requirement from the link above.

Instructions to run the frontend

Copy the GitHub repository (download the zip file / clone the repository, etc.)
Open a terminal and ***cd* to the *frontend/* folder**
Run the command ***npm install*** to install the dependencies specified in *package.json*
While in the *frontend/* folder, run the command ***npm start*** to start the web-app

JavaScript Primer

This should help revise all required concepts in a quick and concise manner
A Re-Introduction to JavaScript

You should also have a decent understanding of HTML, CSS and Bootstrap (or any other CSS library if you are comfortable with that).

Key JavaScript that will be heavily used

let should be used **instead of *var*** in almost all cases. And **use *const*** for objects **that don't take on new values after initialization**.

***forEach* method**

```
array.forEach(value => {
      // code to execute on each element of array
})
```

Adding to an array is typically done using the ***concat*** method, which creates a new array instead of modifying the existing one, i.e.,

```
const m = [1, 2, 3]
const m1 = m.concat(5)
// m still remains the original array
```

***map* Method (VERY IMPORTANT)**

```
arr.map((element) => {return expression that evaluates to new value for
"element"})
// Returns a new array where each element has been applied the evaluation
expression
Example, arr.map(element => return element * 5)
```

Different syntax for *map* which gets the index of the element as well

```
arr.map((element, index) => {return expression})
```

Destructuring arrays and other structures

```
const M = [1, 2, 3, 4, 5]
const [first, second, ...rest] = M
// first = 1, second = 2, rest = [3, 4, 5]
```

Note: There are different ways destructuring is done depending on the context which you might have to revise

Arrow function (VERY IMPORTANT)

```
const sum = (n1, n2) => {
      return n1 + n2
}
sum(3, 5) // returns 8
```

Shorthand version of this is used typically in the *map* method as can be seen.

You can assign functions to properties in JS objects too but in the context of React it is not of much importance because we have React Hooks.

Same for the *class* construct which was introduced in ES6 (ECMAScript 6, standard for JavaScript) and has a lot of usage in React as well but at this point it has essentially become legacy and its usage is not recommended. However you might end up working on these legacy projects so understanding how they're used is also important.

Delete keys from objects using the syntax, *delete object.key*
If the key doesn't exist then no error happens and the operation will return true as it always does.

Creating the React app + Prerequisite React Theory

npx create-react-app frontend

This creates a directory called *frontend* wherever you have run this command.
To start the React app after creating the app, use *npm start* in the new directory that's created.

Delete the following files, *reportWebVitals.js, setupTests.js, logo.svg, App.css, App.test.js*.
These JS files are required for a complete development and testing but not in this case.

Your *index.js* file should look like this after the updates,

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <App />
);
```

All working code will be seen in the *src* folder.
App.js defines all the React components, *index.js* renders the root App component.
The *public* folder essentially is the frontend of the app and it contains the *index.html* file which is the home page.
However all components that will be rendered on *index.html* are defined in *App.js*

React works using components which are written in JSX, looks like HTML but really it's compiling JavaScript at the backend and allows you to easily embed JS variables and expressions along with the HTML code. Embedding must happen with curly braces { }

We are going to be using Functional Components instead of the now legacy Class Components.

You can embed components within a component, like,

```
const App = (props) => (
      <div>
            <p> Hello World </p>
            <Footer name={props.name} />
      </div>
)
```

Here, *Footer* is a component that is embedded within the *App* component. It takes in a property called *name* which itself is given to the App component from the outside. And the *Footer* component will take this as a parameter while being defined, for example,

```
const Footer = ({ name }) => {
  return (
    <footer className='footer rounded text-center p-2'>
      <span>{name}</span>
    </footer>
  )
}
```

Every component file (JS file) must end with the statement,

```
export default ComponentName
```

Otherwise the component will not be accessible outside that JS file.

So components are used for subdividing content into logical subunits to make everything its own module and increase reusability and designing logical components that make sense and making the frontend design more systematic with the help of React.

Rules for components

- Name must start with an uppercase letter
- Typically needs one root element, i.e., a tag that encloses the entire content of that component, in above example, the tag is the root element.
- However this can be bypassed in a couple of ways which you can look up yourself, one typical way is to enclose in an empty tag, like so,

  ```
  <>
        Component JSX
  </>
  ```

  This way you won't have to add an extra root element that is rendered on the HTML web page and everything still looks neat.

Multi-Page React Aps

React is used for Single Page Applications, so no URL routes on the frontend like /payment, /users, etc.
That kind of URL routing is reserved for the backend only.
If you want to enable routing on the frontend and make a multi-page React app then you need another library called React-Router (which we won't need here).

npm

When you create any React app using npm, that is, via *create-react-app*, then a file called *package.json* will be created which gives all the project details like dependencies and scripts to run, etc.

The scripts object in this JSON file will indicate which commands on the terminal will invoke which commands, like *npm start* invokes *react-scripts start.*

So we could add our own commands to this section which will be more useful later on but not for this project.

Note that we can update all the libraries in the current app with the command *npm update*
And the command *npm install* would install all the libraries present in *package.json*. This is useful when we are working on the same project on a different computer.

Version numbers of a library consist of three parts, patch number, minor number and major number. "3.5.2", patch number = 2, minor number = 5, major number = 3

If the *major* number of a dependency does not change, then the newer versions should be backwards compatible. This means that if our application happened to use version 4.99.175 of React in the future, then all the code implemented in this part would still have to work without making changes to the code. In contrast, the future 5.0.0 version of React may contain changes that would cause our application to no longer work.

useState hook

When something is changed on the webpage, the React component has to be rerendered and we have to write explicit code for it. But we can do this in an automated way with the help of React states. So if we attach a state to a component and this state gets updated at any time during the running of the app, the component gets rerendered and we are able to see the updated webpage. For implementing this we have the *useState* hook.

```
const [ counter, setCounter ] = useState(0)
// counter is the stateful variable which is given initial value of 0
// setCounter is the function that we can use to set the value of this variable
```

Whenever *setCounter()* is called, React understands that the state has changed and so it rerenders that component. So **whenever the function that changes the state is called, that component is rerendered.**

Core React design principle is to break down every single thing into as small of a component as possible which can even be reused across projects. Typically, multiple components share the same stateful variables and so the stateful variables should be declared in their closest common ancestor.

Note that the stateful variable can be of any form, doesn't necessarily have to be a single-valued variable. It can even be an object (JSON).

Do not update the state variable directly and only do it via the setter function.
In React it is forbidden to update the state directly on the state variable as it leads to a lot of unforeseen problems (even if the website seems to be working at first glance), instead it must be done by creating a new object, i.e., *counter += 1* is not allowed. Instead you set it using *setCounter(counter + 1)*.

If your state is an object that you want to update, then you must create a new object with the updated state values and then call the setter method of that state with the new object as the parameter like,

```
const [state, setState] = useState({"colour": "green", "name": "yellow"})
// Creating updated state object
const newState = {"colour": "Kero Kero", "name": "Bonito"}
// Actually updating the state
setState(newState)
```

A practical example of a state being used is, have a state called *user* which contains the data of the logged in user. Initially, this will be *null* and upon a successful login we will update the *user* state with the data of the newly logged in user. Because the state changed, the page will be rerendered and we can write logic that will move on from the login screen to the User dashboard screen where user details will be shown.

Rules of React Hooks

Don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function.

Don't call Hooks from regular JavaScript functions. Instead, you can:

- Call Hooks from React function components.
- Call Hooks from custom Hooks

These rules might seem simple but they have complicated implementations at times.

Further Reading: Helpful Resources for Hooks

- Awesome React Hooks Resources
- Easy to understand React Hook recipes by Gabe Ragland
- Why Do React Hooks Rely on Call Order?

axios

It is a library that is used for transferring data to and from the server. Similar to the standardized *fetch()* method but is more pleasant to use and is part of the npm library.

npm install axios

You can fire GET requests using the axios library as, *promiseObj = axios.get(url)*
These requests return a promise.

A Promise is an object representing the eventual completion or failure of an asynchronous operation.
So a promise has 3 states, pending, fulfilled, rejected.

```
promiseObj.then(response => {
        // code here
})
```
is the method to use to access the response of a promise operation.

There's also the *async-await* syntax that can be used in place of promise-chaining which is easier on the eyes and it is what I will be using in the repo, here's an introduction on the two styles.
A Comparison Of async/await Versus then/catch

Effect Hook

The Effect Hook lets you perform side effects in function components. Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. As such, effect hooks are precisely the right tool to use when fetching data from a server.

```
import { useEffect } from 'react'
```

The Effect hook is used as follows,

```
useEffect(() => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}, [])
```

Note that the Effect Hook takes two parameters, first is the function to be executed and the second determines how often the effect is run.
By default it is run on every render, but it is not always necessary and would probably be overkill.
By specifying it as an empty array, i.e. [ ] as above, it is only run on the first render.
If you specify the second parameter as some array of values, then the Effect hook is run whenever any value from the given array changes.

An easy example to map this to is,

```
// Effect Hook that fetches a user's bills
// If the "user" state changes, then that user's bills must be fetched.
// This is why "user" is part of the dependency array of this hook
useEffect(() => {
  // To ensure that the hook doesn't execute when user state gets updated to null
  if (user) {
    axios
    .get('http://localhost:3001/api/bills?user='+user.id)
    .then(billsData => {
      setBills(billsData)
    })
   }
  }, [user])
```

There's many more hooks that exist and are frequently used but this should be enough for the project.

Styling

Add the *<link>* and *<script>* tags mentioned in the below link in the *<head>* tag of the *index.html* file under the *public* folder of your React app.
Download · Bootstrap v5.2

This is what is to be copied as per current versions,

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css"
rel="stylesheet"
integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-OERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJuaOe923+mo//f6V8Qbsw3"
crossorigin="anonymous"></script>
```

Here's other helpful resources for styling,
Get Started with Font Awesome - Free Icons for your page
Google Fonts - Free fonts for your page

And for actually writing up the Bootstrap code for your styles, you should use templates as that'll make things faster and the overall output would also look better.

For example, here's templates for login pages
Bootstrap Login Form - free examples, templates & tutorial

Handling Forms

Final thing that's required is how to handle forms, i.e. how to store input data, how to submit the stored data, etc.

Here, we'll be using the Controlled Components method to do this.
How to Handle Forms in React with Controlled Components

A very brief summary of this technique,
To create forms you must add an *onSubmit* event handler whose first line is the *event.preventDefault()* method call to prevent default submission to action attribute value.

To track the state of input elements in the form, you can create a state variable for each input. Like a state variable for tracking and setting the value typed into a text box input (this is adjusted via the *value* attribute). So in this case, the state variable will be used for rendering the actually typed value by the user into the text box and once the form is submitted, the *onSubmit* event handler will be able to access this value.

Directory Structure for React apps

All our work will be isolated to the *src* directory

App.js defines all the React components, *Index.js* renders the root App component.

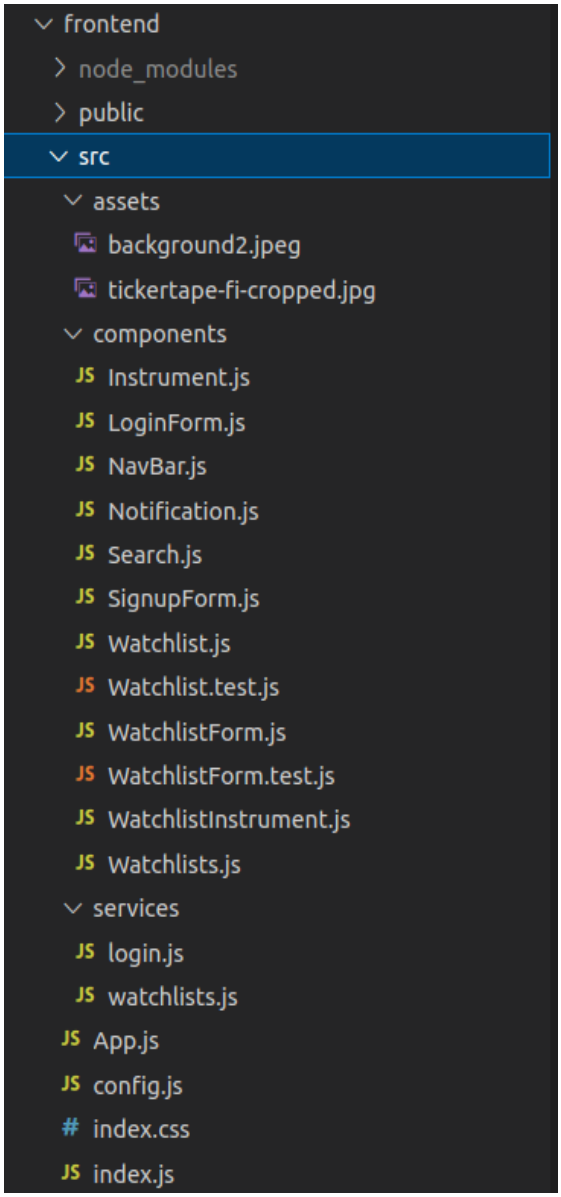
index.css would contain the primary styling of the app. Can be more detailed and structured if you want to make your styling more organized but for most purposes a single css file is alright.

The *assets* folder would typically contain things like images (for the background typically), videos, etc.

The *components* folder would contain the React components which you have designed for your app.

The *services* folder would contain requests related to each entity in your backend. It is a wrapper in order to isolate the API interaction from the UI and rendering aspects. In our use case, the two entities are users and bills, so we have two services, one related to *users* (login user) and the other one related to *bills* (get bills of a user, pay bill).

Here's what a finalized directory structure could look like,



The article [The 100% correct way to structure a React app (or why there's no such thing)] provides some perspective on the issue.

Full Integration Flow

User clicks "Pay Bill" Button which is taken up by the *payBill* onClick event handler.
Here *bill* is the object that is to be rendered.

```
<button onClick={payBill(bill)}>
  Pay Bill
</button>
```

The *payBill* onClick event handler calls the axios *billService payBill* method which is in the *services* folder,

```
const payBill = async (billObject) => {
      try {
      // Call payBill() at the backend
      await billService.payBill(billObject)
      ...
```

The axios payBill method fires a DELETE request to the backend as,

```
const response = await axios.delete(`${billsUrl}/${bill.billId}`)
```

where,

```
const billsUrl = `http://localhost:8081/api/bill`
```

The DELETE request is then passed on to the backend server and received by the *BillController* class which hands it off to the *payBill* method because it is responsible for handling the requests at */bill/{billId}*

```
@DELETE
@Path("/{billId}")
@Produces(MediaType.APPLICATION_JSON)
public Response payBill() {
      ...
```

The *payBill* method of the *BillController* class hands off the control to the *payBill* method of the *BillService* class which implements business logic other than the main task which here is paying the bill, these business logic tasks could include things like data type translations, 3rd party data fetching, etc.

BillService class has an instance of the *BillDAOImpl* class which is referenced to via the *BillDAO* interface

```
BillDAOImpl billDAO = new BillDAOImpl();
```

It then calls the *payBill* method of the *BillDAOImpl* class

```
Boolean successfulPayment = billDAO.payBill(billId);
```

The *payBill* method of the *BillDAOImpl* class actually executes the delete operation on the SQL table via Hibernate query (HQL) in a transaction and returns the Boolean status of the result back to the BillService class.

Now the control flows back from the Service class to the Controller which returns the Boolean status with appropriate status code, 204 if the bill was paid successfully or 400 otherwise.

This response is now received by the axios service at this line (this is why asynchronous execution is important),

```
const response = await axios.delete(`${billsUrl}/${bill.billId}`)
```

And it sends the data back to the onClick event handler,

```
return response.data
```

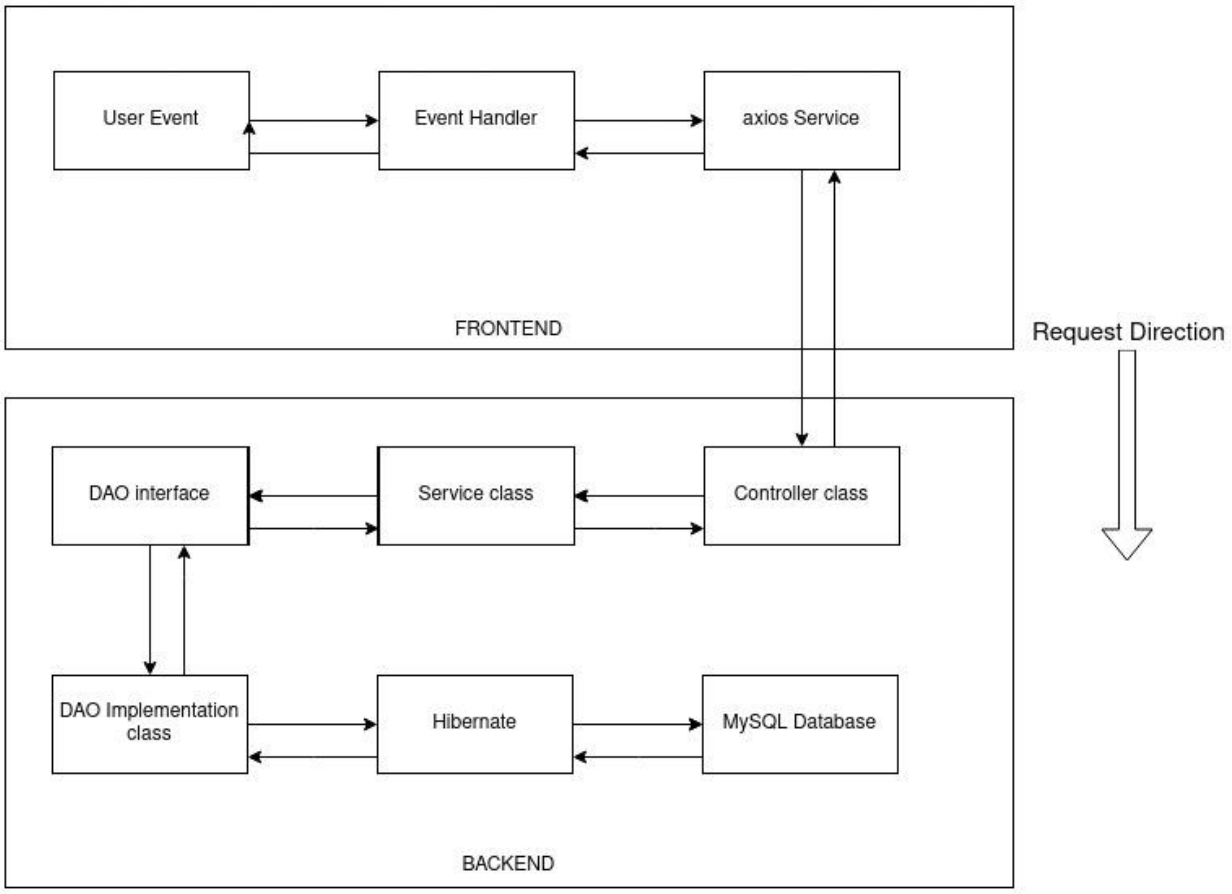
The onClick event handler updates the bills React state to remove the bill that has just been paid.

```
// On successful completion of the above method, iterate through all the bills
// and only retain those bills which don't have ID equal to the billObject's ID,
// i.e. the ID of the bill that's just been paid/deleted
setBills(bills.filter(bill => bill.billId !== billObject.billId))
```

Because the state has just been updated, the *App* component which contains the *bills* state will be re-rendered and upon re-rendering, it will render the bills with the new state value which will no longer contain the bill that was just paid because it was filtered in the above statement. And now the user will see only the remaining bills on the bill dashboard.

The two things that take time in this are the transfer from the frontend to the backend and the database interaction done by Hibernate but everything else happens in the blink of an eye!

Here’s an overview of everything that happened,



Asynchronous Execution of the JavaScript Engine, The Event Loop

This will explain so much about how JavaScript executes and what asynchronous execution is all about. Definitely should check it out even if you have a very basic understanding of JavaScript.

What the heck is the event loop anyway? | Philip Roberts | JSConf EU

Backend

Service Folder/Package

It is the business logic tier of the backend. This is done so that the only thing that's done by the classes in the *controller* package is to receive the request from the frontend, send the request to a service class, receive the response and send this response back to the frontend with the appropriate status code.

REST

Intro

REST is a convention that is followed for designing API URLs at the backend.
APIs are what we write as Controllers at the backend.
Know this distinction well.

According to REST principles (which can vary a lot depending on where you are learning this from but this is what I've learned),
Individual data objects are called resources and every resource has a unique address, i.e., URL associated with it.

Note: For all examples, I'm writing the URLs from the */api/* point onwards only for ease of writing, no domain names like *localhost:8081/api/*

So in our use-case we have,
/api/student/420 => We expect to have student with id 420 at this URL
/api/bill/69 => We expect to have bill with id 69 at this URL

Existing resources are fetched using the HTTP GET request to the appropriate URL.
A new resource is created using the HTTP POST request to the appropriate URL.
This would depend on the REST conventions followed by that resource collection. Basically the backend devs determine how REST conventions are followed at their end and accordingly the frontend devs would have to adjust.

Most importantly, for requests made by the frontend, the *Content-Type* header has to be specified to the appropriate type, in most cases, *axios* will automatically realize this type and fill it appropriately.

Note you don't have to specify a resource URL for POST requests, i.e., ID for that object in the URL because the resource is actually being created now, which is not the case for a PUT

request. So you don't make a POST request to */api/student/1* because this would mean that a student with id 1 already exists, so how are you making another student with id 1?

The PUT HTTP request is used to replace an existing resource by sending the request to the appropriate URL, basically PUT is like the Update HTTP method.

And the PATCH HTTP request is used to modify only some of the properties of an existing object resource. This seems similar to PUT but PUT is the one that's more frequently used as compared to PATCH.

The DELETE HTTP request is used to delete the specified resource.

URL Parameters

There's two types of URL Parameters that exist,

- Path Parameter
- Query Parameter

Path Parameter

An example of Path Parameter REST Controller is the following under the *BillController* class,

```
@DELETE
@Path("/{billId}")
@Produces(MediaType.TEXT_PLAIN)
public Response deleteBill(@PathParam("billId") Integer billId) {
```

The path to this DELETE method is, */api/bill/{billId}* and the meaning is pretty clear in simple English, "Delete the bill whose bill id is specified in the URL"
So for example, a DELETE request to */api/bill/42* would delete the bill with id 42 from the database.

So *{billId}* is the path parameter in this URL.

Query Parameter

An example of Query Parameter REST Controller is the following under the *BillController* class,

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getBills(@QueryParam("studentId") int s_id) {
```

This API basically tells you in simple English, "Get me all the Bills of the student whose id is specified in the URL".
And the path to this GET method is, */api/bills?studentId={studentId}*

This is a bit more complex to understand but this is what that means,
/api/bills?studentId=2
Fetch all the bills of the student with id 2.

You can verify that these URLs are being used from the frontend by checking the *services* folder of the React app.

HTTP Status Codes

Using the correct status code matters a lot when it comes to building proper REST APIs.
The status codes are just a convention and don't really affect execution in any shape or form but it is considered industry standard and helps other developers quickly understand what an API is doing if you follow status code web standards (like 404 for resource not found).

Here's a brief overview of what's generally important,
200 Status Codes represent various forms of successful executions
400 Status Codes represent various forms of bad requests or unauthorized access
500 represents Internal Server Error, so your server has basically crashed or thrown an error at the very least :(

If you want to find what status code you should use for a particular API controller then check out the 200, 300, 400 and 500 series on this page,
HTTP Response Status Codes - developer.mozilla.org

CORS

Stands for Cross-Origin Resource Sharing
[Important for placements 🤑]

The Mozilla documentation page on CORS does a great job at explaining the need of CORS and how it's used.
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
Read up to the "Example of access control scenarios" section atleast.

Here's a video as well if reading text is not your thing,
CORS, Preflight Request, OPTIONS Method | Access Control Allow Origin Error Explained

A brief overview of what CORS is and why we face this issue while developing full-stack web applications,
When the frontend makes a request to the backend (like login POST request) and you don't have a CORS policy set up, the request will be rejected showing a *NS_DOM_BADURI* error on the browser console.
But the request will succeed if you are using something like Postman or other API testing tools.

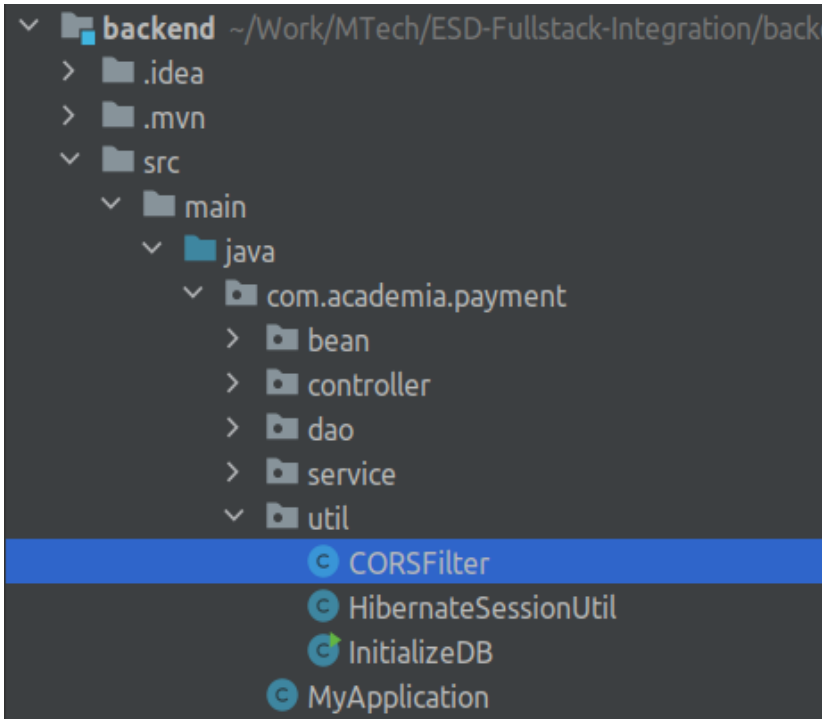
Why does this happen?
Because this request from the frontend to the backend is a Cross-Origin request, i.e. the source and destination are essentially two different domains. Domain name, IP address and even port numbers are enough to make one origin different from the other.
Remember that the React frontend runs on port 3000 and the backend runs on some other port, so even though both are running on your localhost system, this is a cross-origin request.

So in order for this request to succeed, your server must accept the requests that are coming from this domain (http://localhost:3000), and for this we set the Response Header,

```
Access-Control-Allow-Origin: *
```

* = Wildcard character that means the server will accept requests from any domain. This and many other response headers allow you to add a layer of security to your web-app design but that's beyond the scope of anything that we do over here.

This whole thing is done in the Github repo shared by the *CORSFilter* class in the *com.academia.payment.util* package.



And to get this working you also have to make changes to the *MyApplication* class (or *HelloApplication* class in your case) which is the entry point to your API as per the Jersey framework as shown below,

```
@ApplicationPath("/api")
// Note that the class extends ResourceConfig and not Application because
we want to make use of register()
public class MyApplication extends ResourceConfig {
      public MyApplication() {
        // Registering the CORSFilter class with the Jersey ResourceConfig
        register(CORSFilter.class);

        // Telling Jersey the CLASSPATH where the specified classes
        // (in our case, CORSFilter) can be found
        packages("com.academia.payment");
      }
}
```

No need to worry about the technicalities of how this works, just the general idea is enough.

Helpful references for this particular solution with Jersey and CORS:
How to handle CORS using JAX-RS with Jersey - Stack Overflow
Registering Resources and Providers in Jersey 2

Extra features

Persisting Logins

If you want to add the extra feature of persisting logins, i.e. once you have logged in to the website and close it or refresh it without logging out, then on the next opening of the website, you will be automatically logged in. Then you can refer to the below article. I have already implemented this using localStorage in the use case shown but this article is for a more formal look at the whole thing.
How to Persist a Logged-in User in React

Encryption

We are not going to be doing hashing and encryption of passwords or even authorization, but ideally you should be able to implement them using things like JWT.
JSON Web Token Introduction

But this is way too complicated of a feature to implement here but definitely very useful for future learning.

References:

Must-Do

A Re-Introduction to JavaScript

Comparison Of async/await Versus then/catch

How to Handle Forms in React with Controlled Components

Additional Reading

What the heck is the event loop anyway? | Philip Roberts | JSConf EU
(Should definitely watch ASAP after the JS intro)

Full Stack Open 2022 by the University of Helsinki
Amazingly in-depth free course for Full Stack Development (MERN) which is where I learned all of this from.

The 100% correct way to structure a React app (or why there's no such thing)

Cross-Origin Resource Sharing (CORS) - HTTP | MDN (Mozilla Documentation)

How to Persist a Logged-in User in React

Awesome React Hooks Resources

JSON Web Token Introduction

Easy to understand React Hook recipes by Gabe Ragland

Why Do React Hooks Rely on Call Order?

HTTP Response Status Codes - developer.mozilla.org

CORS, Preflight Request, OPTIONS Method | Access Control Allow Origin Error Explained