public/index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1" />

<title>Kaizntree | Approvals</title>

</head>

<body>

<noscript>You need to enable JavaScript to run this app.</noscript>

<div id="root"></div>

</body>

</html>

src/components/AppContextProvider/index.tsx

import { useRef, useState } from "react"

import { AppContext } from "../../utils/context"

import { AppContextProviderComponent } from "./types"

export const AppContextProvider: AppContextProviderComponent = ({ children }) => {

const cache = useRef(new Map<string, string>())

const [error, setError] = useState<string>("")

return (

<AppContext.Provider value={{ setError, cache }}>

{error ? (

<div className="KaizntreeError">

<h1 className="KaizntreeTextHeading--l">Oops. Application broken</h1>

<div className="KaizntreeBreak--l" />

Error: {error}

</div>

) : (

children

)}

</AppContext.Provider>

)

}

src/components/AppContextProvider/types.ts

import { FunctionComponent, PropsWithChildren } from "react"

export type AppContextProviderComponent = FunctionComponent<PropsWithChildren<unknown>>

src/components/InputCheckbox/index.tsx

import classNames from "classnames"

import { useRef } from "react"

import { InputCheckboxComponent } from "./types"

export const InputCheckbox: InputCheckboxComponent = ({ id, checked = false, disabled, onChange }) => {

const { current: inputId } = useRef(`KaizntreeInputCheckbox-${id}`)

return (

<div className="KaizntreeInputCheckbox--container" data-testid={inputId}>

<label

className={classNames("KaizntreeInputCheckbox--label", {

"KaizntreeInputCheckbox--label-checked": checked,

"KaizntreeInputCheckbox--label-disabled": disabled,

})}

/>

<input

id={inputId}

type="checkbox"

className="KaizntreeInputCheckbox--input"

checked={checked}

disabled={disabled}

onChange={() => onChange(!checked)}

/>

</div>

)

}

src/components/InputCheckbox/types.ts

import { FunctionComponent } from "react"

type InputCheckboxProps = {

id: string | number

checked?: boolean

onChange: (newValue: boolean) => void

disabled?: boolean

}

export type InputCheckboxComponent = FunctionComponent<InputCheckboxProps>

src/components/InputSelect

import Downshift from "downshift"

import { useCallback, useState } from "react"

import classNames from "classnames"

import { DropdownPosition, GetDropdownPositionFn, InputSelectOnChange, InputSelectProps } from "./types"

export function InputSelect<TItem>({

label,

defaultValue,

onChange: consumerOnChange,

items,

parseItem,

isLoading,

loadingLabel,

}: InputSelectProps<TItem>) {

const [selectedValue, setSelectedValue] = useState<TItem | null>(defaultValue ?? null)

const [dropdownPosition, setDropdownPosition] = useState<DropdownPosition>({

top: 0,

left: 0,

})

const onChange = useCallback<InputSelectOnChange<TItem>>(

(selectedItem) => {

if (selectedItem === null) {

return

}

consumerOnChange(selectedItem)

setSelectedValue(selectedItem)

},

[consumerOnChange]

)

return (

<Downshift<TItem>

id="KaizntreeSelect"

onChange={onChange}

selectedItem={selectedValue}

itemToString={(item) => (item ? parseItem(item).label : "")}

>

{({

getItemProps,

getLabelProps,

getMenuProps,

isOpen,

highlightedIndex,

selectedItem,

getToggleButtonProps,

inputValue,

}) => {

const toggleProps = getToggleButtonProps()

const parsedSelectedItem = selectedItem === null ? null : parseItem(selectedItem)

return (

<div className="KaizntreeInputSelect--root">

<label className="KaizntreeText--s KaizntreeText--hushed" {...getLabelProps()}>

{label}

</label>

<div className="KaizntreeBreak--xs" />

<div

className="KaizntreeInputSelect--input"

onClick={(event) => {

setDropdownPosition(getDropdownPosition(event.target))

toggleProps.onClick(event)

}}

>

{inputValue}

</div>

<div

className={classNames("KaizntreeInputSelect--dropdown-container", {

"KaizntreeInputSelect--dropdown-container-opened": isOpen,

})}

{...getMenuProps()}

style={{ top: dropdownPosition.top, left: dropdownPosition.left }}

>

{renderItems()}

</div>

</div>

)

function renderItems() {

if (!isOpen) {

return null

}

if (isLoading) {

return <div className="KaizntreeInputSelect--dropdown-item">{loadingLabel}...</div>

}

if (items.length === 0) {

return <div className="KaizntreeInputSelect--dropdown-item">No items</div>

}

return items.map((item, index) => {

const parsedItem = parseItem(item)

return (

<div

key={parsedItem.value}

{...getItemProps({

key: parsedItem.value,

index,

item,

className: classNames("KaizntreeInputSelect--dropdown-item", {

"KaizntreeInputSelect--dropdown-item-highlighted": highlightedIndex === index,

"KaizntreeInputSelect--dropdown-item-selected":

parsedSelectedItem?.value === parsedItem.value,

}),

})}

>

{parsedItem.label}

</div>

)

})

}

}}

</Downshift>

)

}

const getDropdownPosition: GetDropdownPositionFn = (target) => {

if (target instanceof Element) {

const { top, left } = target.getBoundingClientRect()

const { scrollY } = window

return {

top: scrollY + top + 63,

left,

}

}

return { top: 0, left: 0 }

}

src/components/InputSelect/types.ts

export type InputSelectItem = { label: string; value: string }

export type InputSelectProps<TItem> = {

label: string

defaultValue?: TItem | null

onChange: (value: TItem | null) => void

items: TItem[]

parseItem: (item: TItem) => InputSelectItem

isLoading?: boolean

loadingLabel: string

}

export type DropdownPosition = {

top: number

left: number

}

export type InputSelectOnChange<TItem> = (selectedItem: TItem | null) => void

export type GetDropdownPositionFn = (target: EventTarget) => DropdownPosition

src/components/Transactions/TransactionPane.tsx  
import { useState } from "react"

import { InputCheckbox } from "../InputCheckbox"

import { TransactionPaneComponent } from "./types"

export const TransactionPane: TransactionPaneComponent = ({

transaction,

loading,

setTransactionApproval: consumerSetTransactionApproval,

}) => {

const [approved, setApproved] = useState(transaction.approved)

return (

<div className="KaizntreePane">

<div className="KaizntreePane--content">

<p className="KaizntreeText">{transaction.merchant} </p>

<b>{moneyFormatter.format(transaction.amount)}</b>

<p className="KaizntreeText--hushed KaizntreeText--s">

{transaction.employee.firstName} {transaction.employee.lastName} - {transaction.date}

</p>

</div>

<InputCheckbox

id={transaction.id}

checked={approved}

disabled={loading}

onChange={async (newValue) => {

await consumerSetTransactionApproval({

transactionId: transaction.id,

newValue,

})

setApproved(newValue)

}}

/>

</div>

)

}

const moneyFormatter = new Intl.NumberFormat("en-US", {

style: "currency",

currency: "USD",

})

src/components/Transactions/index.tsx

import { useCallback } from "react"

import { useCustomFetch } from "src/hooks/useCustomFetch"

import { SetTransactionApprovalParams } from "src/utils/types"

import { TransactionPane } from "./TransactionPane"

import { SetTransactionApprovalFunction, TransactionsComponent } from "./types"

export const Transactions: TransactionsComponent = ({ transactions }) => {

const { fetchWithoutCache, loading } = useCustomFetch()

const setTransactionApproval = useCallback<SetTransactionApprovalFunction>(

async ({ transactionId, newValue }) => {

await fetchWithoutCache<void, SetTransactionApprovalParams>("setTransactionApproval", {

transactionId,

value: newValue,

})

},

[fetchWithoutCache]

)

if (transactions === null) {

return <div className="KaizntreeLoading--container">Loading...</div>

}

return (

<div data-testid="transaction-container">

{transactions.map((transaction) => (

<TransactionPane

key={transaction.id}

transaction={transaction}

loading={loading}

setTransactionApproval={setTransactionApproval}

/>

))}

</div>

)

}

src/components/Transactions/types.ts

import { FunctionComponent } from "react"

import { Transaction } from "../../utils/types"

export type SetTransactionApprovalFunction = (params: {

transactionId: string

newValue: boolean

}) => Promise<void>

type TransactionsProps = { transactions: Transaction[] | null }

type TransactionPaneProps = {

transaction: Transaction

loading: boolean

approved?: boolean

setTransactionApproval: SetTransactionApprovalFunction

}

export type TransactionsComponent = FunctionComponent<TransactionsProps>

export type TransactionPaneComponent = FunctionComponent<TransactionPaneProps>

src/components/Instructions.tsx

import { Fragment } from "react"

export function Instructions() {

return (

<Fragment>

<h1 className="KaizntreeTextHeading--l">Approve transactions</h1>

<div className="KaizntreeBreak--l" />

<p className="KaizntreeText">

Your company uses Kaizntree as their main financial instrument. You are a manager and you need to

approve the transactions made by your employees.

<span className="KaizntreeBreak--s" />

Select the checkbox on the right to approve or decline the transactions. You can filter

transactions by employee.

</p>

</Fragment>

)

}

src/components/Instructions.tsx

import { Fragment } from "react"

export function Instructions() {

return (

<Fragment>

<h1 className="KaizntreeTextHeading--l">Approve transactions</h1>

<div className="KaizntreeBreak--l" />

<p className="KaizntreeText">

Your company uses Kaizntree as their main financial instrument. You are a manager and you need to

approve the transactions made by your employees.

<span className="KaizntreeBreak--s" />

Select the checkbox on the right to approve or decline the transactions. You can filter

transactions by employee.

</p>

</Fragment>

)

}

src/hooks/types.ts

import { Employee, PaginatedResponse, Transaction } from "../utils/types"

type UseTypeBaseResult<TValue> = {

data: TValue

loading: boolean

invalidateData: () => void

}

type UseTypeBaseAllResult<TValue> = UseTypeBaseResult<TValue> & {

fetchAll: () => Promise<void>

}

type UseTypeBaseByIdResult<TValue> = UseTypeBaseResult<TValue> & {

fetchById: (id: string) => Promise<void>

}

export type EmployeeResult = UseTypeBaseAllResult<Employee[] | null>

export type PaginatedTransactionsResult = UseTypeBaseAllResult<PaginatedResponse<Transaction[]> | null>

export type TransactionsByEmployeeResult = UseTypeBaseByIdResult<Transaction[] | null>

import { useCallback, useContext } from "react"

import { AppContext } from "../utils/context"

import { fakeFetch, RegisteredEndpoints } from "../utils/fetch"

import { useWrappedRequest } from "./useWrappedRequest"

export function useCustomFetch() {

const { cache } = useContext(AppContext)

const { loading, wrappedRequest } = useWrappedRequest()

const fetchWithCache = useCallback(

async <TData, TParams extends object = object>(

endpoint: RegisteredEndpoints,

params?: TParams

): Promise<TData | null> =>

wrappedRequest<TData>(async () => {

const cacheKey = getCacheKey(endpoint, params)

const cacheResponse = cache?.current.get(cacheKey)

if (cacheResponse) {

const data = JSON.parse(cacheResponse)

return data as Promise<TData>

}

const result = await fakeFetch<TData>(endpoint, params)

cache?.current.set(cacheKey, JSON.stringify(result))

return result

}),

[cache, wrappedRequest]

)

const fetchWithoutCache = useCallback(

async <TData, TParams extends object = object>(

endpoint: RegisteredEndpoints,

params?: TParams

): Promise<TData | null> =>

wrappedRequest<TData>(async () => {

const result = await fakeFetch<TData>(endpoint, params)

return result

}),

[wrappedRequest]

)

const clearCache = useCallback(() => {

if (cache?.current === undefined) {

return

}

cache.current = new Map<string, string>()

}, [cache])

const clearCacheByEndpoint = useCallback(

(endpointsToClear: RegisteredEndpoints[]) => {

if (cache?.current === undefined) {

return

}

const cacheKeys = Array.from(cache.current.keys())

for (const key of cacheKeys) {

const clearKey = endpointsToClear.some((endpoint) => key.startsWith(endpoint))

if (clearKey) {

cache.current.delete(key)

}

}

},

[cache]

)

return { fetchWithCache, fetchWithoutCache, clearCache, clearCacheByEndpoint, loading }

}

function getCacheKey(endpoint: RegisteredEndpoints, params?: object) {

return `${endpoint}${params ? `@${JSON.stringify(params)}` : ""}`

}

src/hooks/useCustomFetch.ts

import { useCallback, useContext } from "react"

import { AppContext } from "../utils/context"

import { fakeFetch, RegisteredEndpoints } from "../utils/fetch"

import { useWrappedRequest } from "./useWrappedRequest"

export function useCustomFetch() {

const { cache } = useContext(AppContext)

const { loading, wrappedRequest } = useWrappedRequest()

const fetchWithCache = useCallback(

async <TData, TParams extends object = object>(

endpoint: RegisteredEndpoints,

params?: TParams

): Promise<TData | null> =>

wrappedRequest<TData>(async () => {

const cacheKey = getCacheKey(endpoint, params)

const cacheResponse = cache?.current.get(cacheKey)

if (cacheResponse) {

const data = JSON.parse(cacheResponse)

return data as Promise<TData>

}

const result = await fakeFetch<TData>(endpoint, params)

cache?.current.set(cacheKey, JSON.stringify(result))

return result

}),

[cache, wrappedRequest]

)

const fetchWithoutCache = useCallback(

async <TData, TParams extends object = object>(

endpoint: RegisteredEndpoints,

params?: TParams

): Promise<TData | null> =>

wrappedRequest<TData>(async () => {

const result = await fakeFetch<TData>(endpoint, params)

return result

}),

[wrappedRequest]

)

const clearCache = useCallback(() => {

if (cache?.current === undefined) {

return

}

cache.current = new Map<string, string>()

}, [cache])

const clearCacheByEndpoint = useCallback(

(endpointsToClear: RegisteredEndpoints[]) => {

if (cache?.current === undefined) {

return

}

const cacheKeys = Array.from(cache.current.keys())

for (const key of cacheKeys) {

const clearKey = endpointsToClear.some((endpoint) => key.startsWith(endpoint))

if (clearKey) {

cache.current.delete(key)

}

}

},

[cache]

)

return { fetchWithCache, fetchWithoutCache, clearCache, clearCacheByEndpoint, loading }

}

function getCacheKey(endpoint: RegisteredEndpoints, params?: object) {

return `${endpoint}${params ? `@${JSON.stringify(params)}` : ""}`

}

src/hooks/useEmployees.ts

import { useCallback, useState } from "react"

import { Employee } from "../utils/types"

import { useCustomFetch } from "./useCustomFetch"

import { EmployeeResult } from "./types"

export function useEmployees(): EmployeeResult {

const { fetchWithCache, loading } = useCustomFetch()

const [employees, setEmployees] = useState<Employee[] | null>(null)

const fetchAll = useCallback(async () => {

const employeesData = await fetchWithCache<Employee[]>("employees")

setEmployees(employeesData)

}, [fetchWithCache])

const invalidateData = useCallback(() => {

setEmployees(null)

}, [])

return { data: employees, loading, fetchAll, invalidateData }

}

src/hooks/usePaginatedTransactions.ts

import { useCallback, useState } from "react"

import { PaginatedRequestParams, PaginatedResponse, Transaction } from "../utils/types"

import { PaginatedTransactionsResult } from "./types"

import { useCustomFetch } from "./useCustomFetch"

export function usePaginatedTransactions(): PaginatedTransactionsResult {

const { fetchWithCache, loading } = useCustomFetch()

const [paginatedTransactions, setPaginatedTransactions] = useState<PaginatedResponse<

Transaction[]

> | null>(null)

const fetchAll = useCallback(async () => {

const response = await fetchWithCache<PaginatedResponse<Transaction[]>, PaginatedRequestParams>(

"paginatedTransactions",

{

page: paginatedTransactions === null ? 0 : paginatedTransactions.nextPage,

}

)

setPaginatedTransactions((previousResponse) => {

if (response === null || previousResponse === null) {

return response

}

return { data: response.data, nextPage: response.nextPage }

})

}, [fetchWithCache, paginatedTransactions])

const invalidateData = useCallback(() => {

setPaginatedTransactions(null)

}, [])

return { data: paginatedTransactions, loading, fetchAll, invalidateData }

}

src/hooks/useTransactionsByEmployee.ts

import { useCallback, useState } from "react"

import { RequestByEmployeeParams, Transaction } from "../utils/types"

import { TransactionsByEmployeeResult } from "./types"

import { useCustomFetch } from "./useCustomFetch"

export function useTransactionsByEmployee(): TransactionsByEmployeeResult {

const { fetchWithCache, loading } = useCustomFetch()

const [transactionsByEmployee, setTransactionsByEmployee] = useState<Transaction[] | null>(null)

const fetchById = useCallback(

async (employeeId: string) => {

const data = await fetchWithCache<Transaction[], RequestByEmployeeParams>(

"transactionsByEmployee",

{

employeeId,

}

)

setTransactionsByEmployee(data)

},

[fetchWithCache]

)

const invalidateData = useCallback(() => {

setTransactionsByEmployee(null)

}, [])

return { data: transactionsByEmployee, loading, fetchById, invalidateData }

}

src/hooks/useWrappedRequest.ts

import { useCallback, useContext, useState } from "react"

import { AppContext } from "../utils/context"

export function useWrappedRequest() {

const [loading, setLoading] = useState(false)

const { setError } = useContext(AppContext)

const wrappedRequest = useCallback(

async <TData extends any = void>(promise: () => Promise<TData>): Promise<TData | null> => {

try {

setLoading(true)

const result = await promise()

return result

} catch (error) {

setError(error as string)

return null

} finally {

setLoading(false)

}

},

[setError]

)

return { loading, wrappedRequest }

}

src/utils/constants.ts

import { Employee } from "./types"

export const EMPTY\_EMPLOYEE: Employee = {

id: "",

firstName: "All",

lastName: "Employees",

}

src/utils/context.ts

import { createContext } from "react"

export const AppContext = createContext<AppContextProps>({ setError: () => {} })

type AppContextProps = {

setError: (error: string) => void

cache?: React.MutableRefObject<Map<string, string>>

}

src/utils/fetch.ts

import {

getEmployees,

getTransactionsPaginated,

getTransactionsByEmployee,

setTransactionApproval,

} from "./requests"

import { PaginatedRequestParams, RequestByEmployeeParams, SetTransactionApprovalParams } from "./types"

const timeout = getTimeout()

const mockTimeout = 1 \* timeout

export function fakeFetch<TData, TParams extends object = object>(

endpoint: RegisteredEndpoints,

params?: TParams

): Promise<TData> {

return new Promise((resolve, reject) => {

mockApiLogger({

message: "Loading request",

data: { endpoint, params },

type: "info",

})

let result: TData

try {

switch (endpoint) {

case "employees":

result = getEmployees() as unknown as TData

setTimeout(() => {

mockApiLogger({ data: { endpoint, params, result } })

resolve(result)

}, mockTimeout)

break

case "paginatedTransactions":

result = getTransactionsPaginated(params as PaginatedRequestParams) as unknown as TData

setTimeout(() => {

mockApiLogger({ data: { endpoint, params, result } })

resolve(result)

}, mockTimeout \* 2.5)

break

case "transactionsByEmployee":

result = getTransactionsByEmployee(params as RequestByEmployeeParams) as unknown as TData

setTimeout(() => {

mockApiLogger({ data: { endpoint, params, result } })

resolve(result)

}, mockTimeout \* 1.5)

break

case "setTransactionApproval":

result = setTransactionApproval(params as SetTransactionApprovalParams) as unknown as TData

setTimeout(() => {

mockApiLogger({ data: { endpoint, params, result } })

resolve(result)

}, mockTimeout \* 1)

break

default:

throw new Error("Invalid endpoint")

}

} catch (error) {

if (error instanceof Error) {

mockApiLogger({

message: error.message,

data: { endpoint, params },

type: "error",

})

reject(error.message)

}

}

})

}

function mockApiLogger({

data,

message = "Success request",

type = "success",

}: {

message?: string

data: object

type?: "success" | "error" | "info"

}) {

if (process.env.REACT\_APP\_MOCK\_REQUEST\_LOGS\_ENABLED === "false") {

return

}

console.log(`%c--Fake Request Debugger-- %c${message}`, "color: #717171", getTitleColor())

console.log(data)

function getTitleColor() {

if (type === "error") {

return "color: #d93e3e;"

}

if (type === "info") {

return "color: #1670d2;"

}

return "color: #548a54;"

}

}

function getTimeout() {

const timeout = parseInt(

new URL(document.location as unknown as URL).searchParams.get("timeout") ??

process.env.REACT\_APP\_TIMEOUT\_MULTIPLIER ??

"1000"

)

if (Number.isNaN(timeout)) {

return 1000

}

return timeout

}

export type RegisteredEndpoints =

| "employees"

| "paginatedTransactions"

| "transactionsByEmployee"

| "setTransactionApproval"

src/utils/requests.ts

import {

PaginatedRequestParams,

PaginatedResponse,

RequestByEmployeeParams,

SetTransactionApprovalParams,

Transaction,

Employee,

} from "./types"

import mockData from "../mock-data.json"

const TRANSACTIONS\_PER\_PAGE = 5

const data: { employees: Employee[]; transactions: Transaction[] } = {

employees: mockData.employees,

transactions: mockData.transactions,

}

export const getEmployees = (): Employee[] => data.employees

export const getTransactionsPaginated = ({

page,

}: PaginatedRequestParams): PaginatedResponse<Transaction[]> => {

if (page === null) {

throw new Error("Page cannot be null")

}

const start = page \* TRANSACTIONS\_PER\_PAGE

const end = start + TRANSACTIONS\_PER\_PAGE

if (start > data.transactions.length) {

throw new Error(`Invalid page ${page}`)

}

const nextPage = end < data.transactions.length ? page + 1 : null

return {

nextPage,

data: data.transactions.slice(start, end),

}

}

export const getTransactionsByEmployee = ({ employeeId }: RequestByEmployeeParams) => {

if (!employeeId) {

throw new Error("Employee id cannot be empty")

}

return data.transactions.filter((transaction) => transaction.employee.id === employeeId)

}

export const setTransactionApproval = ({ transactionId, value }: SetTransactionApprovalParams): void => {

const transaction = data.transactions.find(

(currentTransaction) => currentTransaction.id === transactionId

)

if (!transaction) {

throw new Error("Invalid transaction to approve")

}

transaction.approved = value

}

src/utils/types.ts

export type Transaction = {

id: string

amount: number

employee: Employee

merchant: string

date: string

approved: boolean

}

export type Employee = {

id: string

firstName: string

lastName: string

}

export type PaginatedResponse<TData> = {

data: TData

nextPage: number | null

}

export type PaginatedRequestParams = {

page: number | null

}

export type RequestByEmployeeParams = {

employeeId: string

}

export type SetTransactionApprovalParams = {

transactionId: string

value: boolean

}

src/App.tsx

import { Fragment, useCallback, useEffect, useMemo, useState } from "react"

import { InputSelect } from "./components/InputSelect"

import { Instructions } from "./components/Instructions"

import { Transactions } from "./components/Transactions"

import { useEmployees } from "./hooks/useEmployees"

import { usePaginatedTransactions } from "./hooks/usePaginatedTransactions"

import { useTransactionsByEmployee } from "./hooks/useTransactionsByEmployee"

import { EMPTY\_EMPLOYEE } from "./utils/constants"

import { Employee } from "./utils/types"

export function App() {

const { data: employees, ...employeeUtils } = useEmployees()

const { data: paginatedTransactions, ...paginatedTransactionsUtils } = usePaginatedTransactions()

const { data: transactionsByEmployee, ...transactionsByEmployeeUtils } = useTransactionsByEmployee()

const [isLoading, setIsLoading] = useState(false)

const transactions = useMemo(

() => paginatedTransactions?.data ?? transactionsByEmployee ?? null,

[paginatedTransactions, transactionsByEmployee]

)

const loadAllTransactions = useCallback(async () => {

setIsLoading(true)

transactionsByEmployeeUtils.invalidateData()

await employeeUtils.fetchAll()

await paginatedTransactionsUtils.fetchAll()

setIsLoading(false)

}, [employeeUtils, paginatedTransactionsUtils, transactionsByEmployeeUtils])

const loadTransactionsByEmployee = useCallback(

async (employeeId: string) => {

paginatedTransactionsUtils.invalidateData()

await transactionsByEmployeeUtils.fetchById(employeeId)

},

[paginatedTransactionsUtils, transactionsByEmployeeUtils]

)

useEffect(() => {

if (employees === null && !employeeUtils.loading) {

loadAllTransactions()

}

}, [employeeUtils.loading, employees, loadAllTransactions])

return (

<Fragment>

<main className="MainContainer">

<Instructions />

<hr className="KaizntreeBreak--l" />

<InputSelect<Employee>

isLoading={isLoading}

defaultValue={EMPTY\_EMPLOYEE}

items={employees === null ? [] : [EMPTY\_EMPLOYEE, ...employees]}

label="Filter by employee"

loadingLabel="Loading employees"

parseItem={(item) => ({

value: item.id,

label: `${item.firstName} ${item.lastName}`,

})}

onChange={async (newValue) => {

if (newValue === null) {

return

}

await loadTransactionsByEmployee(newValue.id)

}}

/>

<div className="KaizntreeBreak--l" />

<div className="KaizntreeGrid">

<Transactions transactions={transactions} />

{transactions !== null && (

<button

className="KaizntreeButton"

disabled={paginatedTransactionsUtils.loading}

onClick={async () => {

await loadAllTransactions()

}}

>

View More

</button>

)}

</div>

</main>

</Fragment>

)

}

src/index.css

:root {

--color-empty-shade: #fff;

--color-light-shade: #eaeaea;

--color-lighter-shade: #edebe6;

--color-dark-shade: #6f6f66;

--color-darker-shade: #ecf3d3;

--color-text: #2d2d26;

--color-accent: #f4ff56;

--color-constructive: #109d46;

--color-destructive: #d94b03;

}

html,

body {

padding: 0;

margin: 0;

font-family: -apple-system, BlinkMacSystemFont, Segoe UI, Roboto, Oxygen, Ubuntu, Cantarell, Fira Sans,

Droid Sans, Helvetica Neue, sans-serif;

}

a {

color: inherit;

text-decoration: none;

}

\* {

padding: 0;

margin: 0;

box-sizing: border-box;

}

[class^="KaizntreeText--"] {

line-height: 1.5;

font-size: 1rem;

margin: 0;

}

.KaizntreeText--s {

font-size: 0.75rem;

}

.KaizntreeText--hushed {

color: var(--color-dark-shade);

}

[class^="KaizntreeTextHeading--"] {

margin: 0;

line-height: 1;

}

.KaizntreeTextHeading--l {

font-size: 2.5rem;

}

.KaizntreeGrid {

display: grid;

grid-template-columns: 1fr;

gap: 0.75rem;

width: 100%;

}

.KaizntreeBreak--xs {

height: 0.5rem;

}

.KaizntreeBreak--s {

height: 0.75rem;

}

.KaizntreeBreak--l {

height: 1.75rem;

}

hr.KaizntreeBreak--l {

height: 1px;

width: 100%;

margin: 1.75rem 0;

border: none;

background-color: var(--color-light-shade);

}

.KaizntreeInputSelect--root {

width: 100%;

}

.KaizntreeInputSelect--input {

padding: 1rem 0.75rem;

border: 1px solid var(--color-light-shade);

border-bottom: 1px solid var(--color-text);

cursor: pointer;

}

.KaizntreeInputSelect--dropdown-container {

display: none;

position: fixed;

width: 100%;

max-width: 700px;

border: 1px solid var(--color-darker-shade);

margin-top: 0.5rem;

max-height: 16rem;

overflow: auto;

box-shadow: rgb(0 0 0 / 10%) 0px 0px 1px, rgb(0 0 0 / 13%) 0px 4px 8px;

}

.KaizntreeInputSelect--dropdown-container-opened {

display: block;

}

.KaizntreeInputSelect--dropdown-item {

padding: 1rem 0.75rem;

background-color: var(--color-empty-shade);

border-bottom: 1px solid var(--color-light-shade);

cursor: pointer;

}

.KaizntreeInputSelect--dropdown-item-highlighted {

background-color: var(--color-lighter-shade);

}

.KaizntreeInputSelect--dropdown-item-selected {

font-weight: bold;

}

.KaizntreeInputCheckbox--container {

display: flex;

}

.KaizntreeInputCheckbox--label {

border: 1px solid var(--color-light-shade);

background-color: var(--color-lighter-shade);

display: flex;

justify-content: center;

align-items: center;

padding: 0.125rem;

width: 1.25rem;

height: 1.25rem;

cursor: pointer;

}

.KaizntreeInputCheckbox--label-disabled {

cursor: progress;

}

.KaizntreeInputCheckbox--label-checked:before {

content: " ";

width: 100%;

height: 100%;

border-color: var(--color-constructive);

background-color: var(--color-constructive);

}

.KaizntreeInputCheckbox--input {

display: none;

}

.KaizntreeButton {

border: 1px solid var(--color-light-shade);

background-color: var(--color-light-shade);

padding: 0.5rem;

cursor: pointer;

}

.KaizntreeButton:hover {

background-color: var(--color-lighter-shade);

}

.KaizntreeButton:disabled {

cursor: not-allowed;

}

.KaizntreePane {

padding: 1rem;

border: 1px solid var(--color-light-shade);

display: flex;

justify-content: space-between;

align-items: center;

gap: 1rem;

}

.KaizntreePane--content {

flex: 1;

}

.KaizntreeLoading--container {

display: flex;

justify-content: center;

}

.KaizntreeError {

margin: 4rem auto;

max-width: 40rem;

border: 1px solid var(--color-destructive);

color: var(--color-destructive);

padding: 2rem;

}

.MainContainer {

max-width: 700px;

margin: 0 auto;

min-height: 100vh;

padding: 4rem 0;

}

.FilterContainer {

display: flex;

justify-content: flex-end;

gap: 0.75rem;

}

src/index.tsx

import ReactDOM from "react-dom/client"

import "./index.css"

import { App } from "./App"

import { AppContextProvider } from "./components/AppContextProvider"

const root = ReactDOM.createRoot(document.getElementById("root") as HTMLElement)

root.render(

<AppContextProvider>

<App />

</AppContextProvider>

)

src/mock-data.json

{

"employees": [

{

"id": "89bd9324-04e0-4cd6-aa27-981508bd219f",

"firstName": "James",

"lastName": "Smith"

},

{

"id": "7eeba422-5717-4026-8e74-e517576e26bd",

"firstName": "Mary",

"lastName": "Miller"

},

{

"id": "6e88529c-6739-4f1b-bea0-02afdd336bb3",

"firstName": "Linda",

"lastName": "Jones"

}

],

"transactions": [

{

"id": "77af111b-4177-4774-af57-36df9053e1df",

"amount": 95.22,

"employee": {

"id": "89bd9324-04e0-4cd6-aa27-981508bd219f",

"firstName": "James",

"lastName": "Smith",

"extras": 2

},

"merchant": "Social Media Ads Inc",

"date": "8/5/2022",

"approved": true

},

]

}

src/react-app-env.d.ts

/// <reference types="react-scripts" />

In **views.py**, **serializers.py**, and **models.py**, you need to adjust the import paths to reflect the new structure. Here's how you can do it:

# backend/api/views.py

from rest\_framework import generics

from src.models import Item, Transaction, Employee

from src.serializers import ItemSerializer, TransactionSerializer, EmployeeSerializer

class ItemListAPIView(generics.ListAPIView):

queryset = Item.objects.all()

serializer\_class = ItemSerializer

class TransactionListAPIView(generics.ListAPIView):

queryset = Transaction.objects.all()

serializer\_class = TransactionSerializer

class EmployeeListAPIView(generics.ListAPIView):

queryset = Employee.objects.all()

serializer\_class = EmployeeSerializer

# backend/api/serializers.py

from rest\_framework import serializers

from src.models import Item, Transaction, Employee

class ItemSerializer(serializers.ModelSerializer):

class Meta:

model = Item

fields = '\_\_all\_\_'

class TransactionSerializer(serializers.ModelSerializer):

class Meta:

model = Transaction

fields = '\_\_all\_\_'

class EmployeeSerializer(serializers.ModelSerializer):

class Meta:

model = Employee

fields = '\_\_all\_\_'

# backend/api/urls.py

from django.urls import path

from . import views

urlpatterns = [

path('items/', views.ItemListAPIView.as\_view(), name='item-list'),

path('transactions/', views.TransactionListAPIView.as\_view(), name='transaction-list'),

path('employees/', views.EmployeeListAPIView.as\_view(), name='employee-list'),

]

# src/urls.py

from django.urls import path, include

urlpatterns = [

path('api/', include('backend.api.urls')),

] # backend/models.py

from django.db import models

class Item(models.Model):

sku = models.CharField(max\_length=100)

name = models.CharField(max\_length=255)

category = models.CharField(max\_length=100)

tags = models.CharField(max\_length=255)

stock\_status = models.CharField(max\_length=50)

available\_stock = models.IntegerField()

class Transaction(models.Model):

amount = models.DecimalField(max\_digits=10, decimal\_places=2)

employee = models.ForeignKey('Employee', on\_delete=models.CASCADE)

merchant = models.CharField(max\_length=255)

date = models.DateField()

approved = models.BooleanField()

class Employee(models.Model):

id = models.UUIDField(primary\_key=True)

firstName = models.CharField(max\_length=100)

lastName = models.CharField(max\_length=100)

extras = models.IntegerField()

# backend/migrations/0001\_initial.py

from django.db import migrations, models

class Migration(migrations.Migration):

initial = True

dependencies = []

operations = [

migrations.CreateModel(

name='Item',

fields=[

('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),

('sku', models.CharField(max\_length=100)),

('name', models.CharField(max\_length=255)),

('category', models.CharField(max\_length=100)),

('tags', models.CharField(max\_length=255)),

('stock\_status', models.CharField(max\_length=50)),

('available\_stock', models.IntegerField()),

],

),

migrations.CreateModel(

name='Transaction',

fields=[

('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),

('amount', models.DecimalField(decimal\_places=2, max\_digits=10)),

('merchant', models.CharField(max\_length=255)),

('date', models.DateField()),

('approved', models.BooleanField()),

('employee', models.ForeignKey(on\_delete=django.db.models.deletion.CASCADE, to='backend.Employee')),

],

),

migrations.CreateModel(

name='Employee',

fields=[

('id', models.UUIDField(primary\_key=True, serialize=False)),

('firstName', models.CharField(max\_length=100)),

('lastName', models.CharField(max\_length=100)),

('extras', models.IntegerField()),

],

),

]