Project Doctor Demand Response Developer Documentation

# Project Overview

# Application Architecture

MERN Stack: MongoDB, ExpressJS, ReactJS, NodeJS

React

Express

Node

Mongo

MVC Design Pattern using MERN

Special Library Calls Used

# Codebase Overview

# Frontend Code Subfolders and Files

## node\_modules folder

## public folder

## src folder

## index.js

This is the first page that loads when going to the site. It renders App.js.

## App.js

This is the main layout page of the site. It is loaded by index.js. It lays out the main page with the dashboard and allows the browser to access special pages like the Signup, Activate, Login, and System Down pages.

## registerServiceWorker.js

# src Folder Subfolders and Files

## assets

## components

## contexts

## demos

## hocs

## pages

## renderers

## services

## styles

## utils

## views

### SignupView.js

**The signup view (in *src/frontend/views/SignupView.js*) renders the app’s signup page where users can sign up for an account. It imports and renders the authForm component (in *src/frontend/views/AuthForm.js*). The authForm component takes all the input entered by the user, validates the data, and handles the subscribe call to the UserService’s** registerUser **function** **(in *src/frontend/services/UserService.js*). The UserService’s** registerUser **function in turn subscribes to the RestService (in *src/frontend/services/RestService.js*) which issues an HTTP** put **request to the backend API. The** put **request contains the “register” parameter and the** userInfo **object which contains the data that was entered into the form. When the backend router (in *src/backend/routes/index.js*) receives the request it sees the “register” parameter and maps the request to the userController’s** registerUser() **function (in *src/backend/controllers/user.controller.js*) which processes the data, and then sends a response containing a** status **field and a** data **object, which contains an** existingUser **field. This response is forwarded back to the frontend UserService where the request originated. Because the AuthForm is subscribing to this service, it “sees” the response and takes action depending on the** status **and** existingUser **fields contained in the response. If** status **is** true **the registration was successful and the user is notified that an email containing an activation code to click on has been sent to them. If** status **is not** true **AND** existingUser **is** true **the user is notified that the account already exists, if neither** status **nor** existingUser **is** true**, the user is notified that the registration has failed.**

**Functions in SignupView.js**

render() {…}

This is the only function in the Signup view. It converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser, including the AuthForm component. The bulk of the page’s functionality is contained in the AuthForm component.

### AuthForm.js

This is the main component in the SignupView. It handles all the user input fields and button events on the Signup page, and subscribes to the UserService which communicates with the backend.

**Functions in AuthForm.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. These fields include the Organization Name, Address Line 1, Address Line 2, City, State, Zip Code, Password, Email, Alternate Email, Phone, and Alternate Phone input boxes. When the information entered in these input boxes changes, the handleUserInfoChange function takes the new input and updates the state variables in the userInfo state object which will be sent to the backend to be inserted into the database.

handleChange(event) {…}

This function handles changes the user has made to the retyped password field on the form in their browser. It takes the new input and updates the retyped password state variable, which will be used by the validate(af)function to make sure the user has typed their password correctly

updateUserType(event) {…}

This function handles the event that the user clicks on either the Healthcare Provider or Volunteer tab at the top of the form. It updates the type state variable in the UserInfo object, setting it to ‘H’ if the user has clicked on the Healthcare Provider tab and ‘V’ if the user has clicked on the Volunteer tab.

handleTermsChange(event) {…}

This function handles changes the user has made to the Terms and Conditions checkbox on the form in their browser. It takes the new input and updates the terms state variable, which is to either enable or disable the Signup button. The Terms and Conditions checkbox must be checked for the Signup button to be enabled.

redirectToLogin(event) {…}

This function handles the event that the user clicks the Login button on the form. It redirects the user to the LoginView.

registerUser = event => {…}

This function handles the event that the Signup button has been clicked and the form has been submitted. It sets the user\_name field to the value of the email field (The user’s Email is used as the username throughout the app) takes the user’s other input and updates the state variables in the userInfo state object, calls the validate(af)function to validate the user input and print appropriate messages if something has not been entered properly, and then calls the UserService.registerUser(…).subscribe(resp => {…}) function which sends the userInfo object to the backend and awaits a response from the backend regarding whether or not the registration data has been successfully entered into the database.

UserService.registerUser(…).subscribe(resp => {…})

This function is called by the form’s registerUser function. It sends the userInfo object to the UserService’s registerUser function, and subscribes to the USerService so that it will be notified when the UserService gets a response from the backend. The response the UserService gets contains a status field and a data object which contains an existingUser field. The status field tells the function whether or not the registration data has been successfully entered into the database so that the registrationSuccess state variable can be set properly, and the existingUser field tells the function if the user already exists so that an appropriate message can be displayed.

validate(af) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank, that the password is at least 8 characters in length, and that the password and the retyped password match.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### ActivationView.js

**The activation view (in *src/frontend/views/ActivationView.js*) renders the app’s activation page where users who have completed the signup form, received the activation email, and clicked on the activation link, can activate their account. It takes the user’s user\_name and activation code parameters from the url string, and handles the subscribe call to the UserService’s activateUser function (in *src/frontend/services/UserService.js*). The UserService’s activateUser function in turn subscribes to the RestService (in *src/frontend/services/RestService.js*) which sends a post request to the backend API. The post request contains the** **“activate” parameter and the userInfo object which contains the user’s user\_name and activation code . When the backend router (in *src/backend/routes/index.js*) receives the request it sees the “activate” parameter and maps the request to the userController’s activateUser() function (in *src/backend/controllers/user.controller.js*). The userController’s activateUser() function receives and processes the request, and then sends a response. The response is forwarded to the frontend UserService and the information it contains is passed on to the ActivationView. If the** **status field in the response is true then the account has been successfully activated, and the user is redirected to the login page where they can log in to their account for the first time. If the status is not true then the user’s account has not been activated and the user receives an appropriate message.**

**Functions in ActivationView.js**

constructor(props) {…}

Initializes the component’s variables and functions.

componentDidMount()

This function is called as soon as the page loads. It uses the queryString.parse(this.props.location.search) function to get the user\_name and code variables from the browser’s url string. Then it calls the UserService.activateUser({…}).subscribe(resp => {…}function which sends the userInfo object to the backend and awaits a response from the backend regarding whether or not the activation has been processed successfully.

UserService.activateUser({…}).subscribe(resp => {…})

This function is called by the form’s componentDidMount() function. It sends the userInfo object to the UserService’s activateUser function, and subscribes to the USerService so that it will be notified when the UserService gets a response from the backend. The response the UserService gets contains a status field and an accountStatus field which tell the function whether or not the activation has been successful and whether or not the user’s account is suspended. If the activation has been successful the user is redirected to the login page and a url parameter indicating that the user is a new user is sent along with the redirect. If the activation has not been successful, the activationFailed state variable is set to true and the user is notified. If the failure was due to the user’s account being suspended additional information explaining that case is included in the message to the user.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### LoginView.js

**The login view (in *src/frontend/views/LoginView.js*) renders the app’s login page where users whose accounts have been activated can log in. It imports and renders the LoginForm component (in *src/frontend/views/LoginForm.js*). The LoginForm component takes the user name and password entered by the user, validates the data, and handles the subscribe call to the AuthenticationService’s** login **function (in *src/frontend/services/AuthenticationService.js*). The AuthenticationService’s** login **function in turn subscribes to the RestService (in *src/frontend/services/RestService.js*) which issues a** post **request to the backend API. The** post **request contains the “users/login” parameter and the** loginInfo **object which contains the data that was entered into the form. When the backend router (in *src/backend/routes/users.js*) receives the request it sees the “register” parameter and maps the request to the userController’s** login **function (in *src/backend/controllers/user.controller.js*) which processes the data, and then sends a response containing a** status **field and a** data **object, which contains** userNotActivated,userSuspended**,** loginFailed**, and** existingUser **fields. This response is forwarded back to the frontend UserService where the request originated. Because the LoginForm is subscribing to this service, it “sees” the response and takes action depending on the response’s** status **field and the fields contained in the** data **object. If** status **is** true **the login was successful and the user is redirected to their home page. If** status **is not** true **the login has failed, and the user is notified of the reason why the login failed. The login can fail if the user does not exist, if the user exists but is not activated, if the user has entered an incorrect password, or if the user is suspended. If the user tries to login unsuccessfully more than five times their account is suspended, and they must click on “Forgot Password” to request a password reset.**

**Functions in** **LoginView .js**

render() {…}

This is the only function in the Login view. It converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser, including the LoginForm component. The bulk of the page’s functionality is contained in the LoginForm component.

### LoginForm.js

This is the main component in the LoginView. It handles all the user input fields and button events on the login page, and subscribes to the AuthenticationService which communicates with the backend.

**Functions in** **LoginForm.js**

constructor(props) {…}

Initializes the component’s variables and functions.

componentDidMount()

This function is called as soon as the page loads.

handleLoginInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

redirectToSignup(event) {…}

This function handles the event that the user clicks the Signup button on the form. It redirects the user to the SignupView.

redirectToUpdatePasswordRequest(event) {…}

This function handles the event that the user clicks the Forgot Password button on the form. It redirects the user to the updatePasswordRequestView.

handleChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

login = event => {…}

This function handles the event that the Login button has been clicked and the form has been submitted. It takes the user’s input and updates the state variables in the loginInfo state object, calls the validate(lf)function to validate the user input and print appropriate messages if something has not been entered properly, and then calls the AuthenticationService.login(…).subscribe(resp => {…}) function which sends the loginInfo object to the backend and awaits a response from the backend regarding whether or not the user has been successfully logged in.

AuthenticationService.login(…).subscribe(resp => {…})

This function is called by the form’s login = event => {…} function. It sends the loginInfo object to the AuthenticationService’s login function, and subscribes to the AuthenticationService so that it will be notified when the AuthenticationService gets a response from the backend. The response the AuthenticationService gets **a** status **field and a** data **object, which contains** userNotActivated,userSuspended**,** loginFailed**, and** existingUser **fields**. The status **field** tells the function whether or not the login has been successful. If the login has been successful the user is redirected to their home page. If the login has not been successful, the function examines the four fields contained in the response’s data object and tells the user why the login was unsuccessful according to their particular case. There are 5 reasons why ogins can be unsuccessful: 1) the user does not exist, in which case the user is told to click the signup button and register 2) the user has entered the wrong password, in which case they are told to try again 3) the user exists but their account is not activated, in which case the user is told to check their email for the activation email which should have been sent when they registered 4) the user’s account has been suspended, in which case they are told to click Forgot Password to request a password reset

validate(lf) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains into HTML/JS code to be rendered in the user’s browser

### HomeView.js

**The home view (in *src/frontend/views/HomeView.js*) renders the app’s home page where logged in users can view their dashboard. It renders the user’s Dashboard and imports and renders the MapWithBubbles (in *src/frontend/components/MapWithBubbles.js*), Page (in *src/frontend/components/Page.js*),, SupportTicket (in *src/frontend/components/SupportTicket.js*), and IconWidget (in *src/frontend/components/Widgets/IconWidget.js*) components. It also imports the SupportTicketsData object (in *src/frontend/demos/dashboardPage.js*). The MapWithBubbles component is used for the map that appears on the home page . The SupportTicket component is used to display. The IconWidget is used for the icons at the top of the home page displaying the numbers of healthcare providers, volunteers, supply requests, and the inventory count. The SupportTicketsData object . The LoginForm component takes the user name and password entered by the user, validates the data, and handles the subscribe call to the AuthenticationService’s** login **function (in *src/frontend/services/AuthenticationService.js*). The AuthenticationService’s** login **function in turn subscribes to the RestService (in *src/frontend/services/RestService.js*) which issues a** post **request to the backend API. The** post **request contains the “users/login” parameter and the** loginInfo **object which contains the data that was entered into the form. When the backend router (in *src/backend/routes/users.js*) receives the request it sees the “register” parameter and maps the request to the userController’s** login **function (in *src/backend/controllers/user.controller.js*) which processes the data, and then sends a response containing a** status **field and a** data **object,**

**Functions in HomeView.js**

constructor(props) {…}

Initializes the component’s variables and functions.

componentDidMount() {…}

DashboardService.getDashboardData().subscribe(resp => {…})

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser.

### RequestAndInventoryView.js

**The request and inventory view (in *src/frontend/views/RequestAndInventoryView.js*) renders the app’s request and inventory page where logged in users can view the supply requests from organizations and the inventories of makers. It imports and renders the RequestInventoryContext.Provider (in *src/frontend/contexts/RequestInventoryContext.js*), SupplyRequestForm (in *src/frontend/views/SupplyRequestForm.js*), InventoryForm (in *src/frontend/views/InventoryForm.js*), SupplyRequestDetailsModal (in *src/frontend/views/SupplyRequestDetailsModal.js*), and InventoryDetailsModal (in *src/frontend/views/InventoryDetailsModal.js*) components. The RequestInventoryCOntext.Provider… The SupplyRequestForm allows organizations to enter requests for supplies of Masks, Gloves, Face Shields, Gowns, and Other items The InventoryForm allows volunteers to enter their inventory of these supplies. The SupplyRequestDetailsModal and the InventoryDetailsModal**

**Functions in** **RequestAndInventoryView.js**

constructor(props) {…}

Initializes the component’s variables and functions.

componentDidMount()

window.scrollTo(0, 0);

This is needed, because InfiniteCalendar forces window scroll.

this.state.srRefreshEvent.subscribe(() => {this.refreshSRGrid();});

this.state.inventoryRefreshEvent.subscribe(() => {this.refreshInventoryGrid();});

onSRGridReady(params) {…}

this.setState({...this.state, supplyRequestTableApi: params.api})

params.api.setDatasource(…)

this.supplyRequestDataSource()

params.api.sizeColumnsToFit();

refreshSRGrid() {…}

this.state.supplyRequestTableApi.purgeInfiniteCache();

refreshInventoryGrid() {…}

this.state.inventoryTableApi.purgeInfiniteCache();

onInventoryGridReady(params) {…}

this.setState({...this.state, inventoryTableApi: params.api});

params.api.setDatasource(this.inventoryDataSource());

params.api.sizeColumnsToFit();

supplyRequestModal(isNew) {…}

this.setState({...this.state, isNewSupplyRequest: isNew,

supplyRequestModal: true});

inventoryModal(isNew) {…}

this.setState({...this.state, isNewInventory: isNew, inventoryModal: true});

toggleSupplyRequestModal = modalType => () => {…}

if (!modalType) {

return this.setState({ ...this.state,

supplyRequestModal: !this.state.supplyRequestModal,

});

}

};

toggleInventoryDetailModal = () => {this.state.inventoryModalEvent.next(this.state.selectedInventory);};

toggleSRDetailModal = () => {this.state.srModalEvent.next(this.state.selectedSupplyRequest);};

toggleInventoryModal= modalType => () => {

if (!modalType) {

return this.setState({ ...this.state,

inventoryModal: !this.state.inventoryModal

});

}

};

supplyRequestDataSource() {

let ds = {

getRows: function(params) {

// Load supply requests

SupplyRequestService.search(GridUtil.transformGridParams(params)).subscribe(resp => {

if(resp.status === true) {

this.setState({...this.state, supplyRequestData: resp.data});

params.successCallback(resp.data, -1);

return this.state.supplyRequestData.data;

}

});

}

};

ds.getRows = ds.getRows.bind(this);

return ds;

}

inventoryDataSource() {

let ds = {

getRows: function(params) {

// Load inventories

InventoryService.search(GridUtil.transformGridParams(params)).subscribe(resp => {

if(resp.status === true) {

this.setState({...this.state, inventoryData: resp.data});

params.successCallback(resp.data, -1);

return this.state.inventoryData.data;

}

});

}

};

ds.getRows = ds.getRows.bind(this);

return ds;

}

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### SupplyRequestDetailsModal.js

**Functions in** **SupplyRequestDetailsModal.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

handleChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

UserService.registerUser(this.state.userInfo).subscribe(resp => {…})

validate(af) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### SupplyRequestForm.js

**Functions in SupplyRequestForm.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

handleChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

UserService.registerUser(this.state.userInfo).subscribe(resp => {…})

validate(af) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### InventoryDetailsModal.js

**Functions in** **InventoryDetailsModal.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

handleChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

UserService.registerUser(this.state.userInfo).subscribe(resp => {…})

validate(af) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### InventoryForm.js

**Functions in** **InventoryForm.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

handleChange(event) {…}

This function handles button events the user has raised by clicking buttons on the form in their browser.

UserService.registerUser(this.state.userInfo).subscribe(resp => {…})

validate(af) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### HcpView.js

### VolunteersView.js

### UpdatePasswordRequestView.js

**The update password request view (in *src/frontend/views/UpdatePasswordRequestView.js*) renders the app’s password update page where active or suspended users can put in a request to update their password. It imports and renders the UpdatePasswordRequestForm component (in *src/frontend/views/UpdatePasswordRequestForm.js*). The UpdatePasswordRequestForm component takes the** user name **entered by the user, and handles the subscribe call to the UserService’s** updatePasswordRequest **function (in *src/frontend/services/UserService.js*). The UserService’s** updatePasswordRequest **function in turn subscribes to the RestService (in *src/frontend/services/RestService.js*) which issues a** post **request to the backend API. The** post **request contains the “users/updatePasswordRequest” parameter and the** userInfo **object which contains the** user name **that was entered into the form. When the backend router (in *src/backend/routes/users.js*) receives the request it sees the “users/updatePasswordRequest” parameter and maps the request to the userController’s** updatePasswordRequest **function (in *src/backend/controllers/user.controller.js*) which checks is the user exists and if their account has been activated, and generates and emails them a password reset link if they are. Once the** updatePasswordRequest **is done processing, it sends a response containing a** status **field and a** data **object, which contains the** userNotActivated **and** existingUser **fields. This response is forwarded back to the frontend UserService where the request originated. Because the UpdatePasswordRequestForm is subscribing to this service, it “sees” the response and takes action depending on the response’s** status **field and the fields contained in the** data **object. If** status **is** true **the update password request was successful and the user is notified that a password reset link has been sent to their email. If** status **is not** true **the update password request has failed, and the user is notified of the reason why it failed. The update password request can fail if the user does not exist, or if the user exists but is not activated.**

**Functions in UpdatePasswordRequestView.js**

render() {…}

This is the only function in the UpdatePasswordRequest view. It converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser, including the UpdatePasswordRequestForm component. The bulk of the page’s functionality is contained in the UpdatePasswordRequestForm component.

### UpdatePasswordRequestForm.js

This is the main component in the UpdatePasswordRequestView. It handles all the user input fields and button events on the UpdatePasswordRequest page, and subscribes to the UserService which communicates with the backend.

**Functions in** **UpdatePasswordRequestForm.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

handleChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

updatePasswordRequest = event => {}

This function handles the event that the Reset Password button has been clicked and the form has been submitted. It sets the user\_name state variable of the userInfo state object to the value the user has entered into the form, calls the validate(pr)function to validate the user input and print appropriate messages if something has not been entered properly, and then calls the UserService.registerUser(…).subscribe(resp => {…}) function which sends the userInfo object to the backend and awaits a response from the backend regarding whether or not the registration data has been successfully entered into the database.

UserService.updatePasswordRequest(this.state.userInfo).subscribe(resp => {…})

validate(pr) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### UpdatePasswordView.js

**The update password view (in *src/frontend/views/UpdatePasswordView.js*) renders the app’s update password page where users who have submitted an update password request, received the update password email, and clicked on the link contained in the email, can update their password. It imports and renders the UpdatePasswordForm component (in *src/frontend/views/UpdatePasswordForm.js*). The UpdatePasswordForm component takes the user name, the new password, and the retyped new password entered by the user, verifies that the new password and the retyped new password match, and handles the subscribe call to the** **UserService’s** updatePassword **function (in *src/frontend/services/UserService.js*). The UserService’s** updatePassword **function in turn subscribes to the RestService (in *src/frontend/services/RestService.js*) which issues a** post **request to the backend API. The** post **request contains the “users/updatePassword” parameter and the** userInfo **object which contains the** user\_name **and the new** password **that were entered into the form. When the backend router (in *src/backend/routes/users.js*) receives the request it sees the “users/updatePassword” parameter and maps the request to the userController’s** updatePassword **function (in *src/backend/controllers/user.controller.js*) which checks that the user exists and that their account is active, encrypts and stores the password in the database if they are, and then sends a response containing a** status **field and a** data **object, which contains the** userNotActivated **and** existingUser **fields. The** status **field tells the frontend whether or not the update was successful, and the** userNotActivated **and** existingUser **fields, contained in the** data **object, tell the frontend if the user exists and is active This response is forwarded back to the frontend UserService where the request originated. Because the UpdatePasswordForm is subscribing to this service, it “sees” the response and takes action depending on the response’s** status **field and the fields contained in the** data **object. If** status **is** true **the password update was successful and the user is redirected to the login page. If** status **is not** true **the password update has failed, and the user is notified of the reason why the login failed. The password update can fail if the user does not exist, or if the user exists but is not activated.**

**Functions in** **UpdatePasswordView.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

handleChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

UserService.registerUser(this.state.userInfo).subscribe(resp => {…})

validate(af) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### UpdatePasswordForm.js

**Functions in UpdatePasswordForm.js**

constructor(props) {…}

Initializes the component’s variables and functions.

handleUserInfoChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

handleChange(event) {…}

This function handles changes the user has made to the input fields on the form in their browser. It takes the new input and updates the state variables.

UserService.registerUser(this.state.userInfo).subscribe(resp => {…})

validate(af) {…}

This function validates the information that the user has entered into the form. It makes sure that none of the form fields have been left blank.

render() {…}

Converts the RJX code it contains to HTML/JS code to be rendered in the user’s browser

### AdminView.js

## Services

### BaseService,js

### RESTService,js

### AuthService.js

### DashboardService.js

### UserService,js

### InventoryService.js

### SupplyRequestService.js

# Backend Code

## Routes

### index.js

var express = require('express');

This makes the ExpressJS framework required for the code in this file. This is needed to instantiate an ExpressJS router.

var router = express.Router();

This instantiates an ExpressJS router which handles all the forwarding of requests and responses between the backend controllers and the frontend service

const userController = require('../controllers/user.controller.js');

This makes the UserController required for the code in this file. This is required for the router to route requests to the endpoints in the user controller.

var auth = require("../security/auth");

This makes the code in security/auth required for the code in this file. This is needed for requests that require the authentication of user credentials.

// Register Account

router.put('/register', async function(req, res, next) {

let data = await userController.registerUser(req, res);

res.send(data);

});

When a request to “register” is made by the HTTP server the router uses this code to put the request to the registerUser function of the userController (contained in the backend/controllers/user.controller.js file). When it receives a response from the user controller it sends it back to the frontend service which passes the information it contains to the view/form where the request originated. In this case the reponse is simply a status of true or false, true meaning the user was successfully registered and false meaning the user was not successfully registered. The async and await keywords are used to make the processing asynchronous. This means the process won’t keep other processes from running in the case that it takes a long time to run.

// Un-Register Account

router.post('/unregister', auth.authenticate(), function(req, res, next) {

userController.unRegisterUser(req, res);

res.send({status: true});

});

// Activate user

router.post('/activate', async function(req, res, next) {

let status = await userController.activateUser(req, res);

res.send({status: status});

});

module.exports = router;

### dashboard.js

### supply-requests.js

### inventory.js

### user.js

## Controllers

### dashboard.controller.js

### supply-request.controller.js

### inventory.controller.js

### user.controller.js

## Data Access

### db-manager.js

### mongo.dataaccess.js

# Dev Tools and Debugging

## Postman