**CORE FUNCTIONALITIES**

**For Backend:**

**LoginController.java** :The below code is related to login controller where login related endpoints are exposed here. I have shared code snippet of important endpoints like signup, login etc. But this layer contains other endpoints like change password, verify OTP and retry OTP. The implementation of the respective endpoint would be in service layer and database related action would be done in repository layer. Required entities are created for Login changes to interact with tables in database. Custom exception handling is done to throw user defined exceptions. Configuration related to login changes would be available in resources folder. The unit tests are written in src/test/java layer for functional testing and code coverage.

@CrossOrigin(origins = "http://localhost:3000")

@RestController

@RequestMapping("/ppms/user")

**public** **class** LoginController {

@Autowired

LoginService loginService;

@PostMapping("/signup")

**public** ResponseEntity<String> signup(@Valid @RequestBody User user) **throws** UserNameAlreadyExistsException, UserEmailIdAlreadyExistsException{

**try** {

String ppt= loginService.signup(user);

**return** ResponseEntity.*ok*(ppt);

}**catch**(UserNameAlreadyExistsException e) {

**throw** **new** UserNameAlreadyExistsException(e.getMessage());

}**catch**(UserEmailIdAlreadyExistsException e) {

**throw** **new** UserEmailIdAlreadyExistsException(e.getMessage());

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***BAD\_REQUEST***).body(e.getMessage());

}

}

@PostMapping("/login/{username}/{password}")

**public** ResponseEntity<UserData> loginFirstStep(@PathVariable String username, @PathVariable String password) **throws** InvalidLoginCredentialsException, UserNotFoundException{

**try** {

UserData response = loginService.loginUserFirstStep(username, password);

**return** ResponseEntity.*ok*(response);

}**catch**(InvalidLoginCredentialsException e) {

**throw** **new** InvalidLoginCredentialsException(e.getMessage());

}**catch**(UserNotFoundException e) {

**throw** **new** UserNotFoundException(e.getMessage());

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***BAD\_REQUEST***).body(**null**);

}

**LoginController.java** - The below code is related to login controller where otp related endpoints are exposed here. I have shared code snippet of important endpoints like verify OTP and change password etc. The implementation of the respective endpoint would be in service layer and database related action would be done in repository layer. Required entities are created for Login changes to interact with tables in database. Custom exception handling is done to throw user defined exceptions. Configuration related to login changes would be available in resources folder. The unit tests are written in src/test/java layer for functional testing and code coverage.

@PostMapping("/login/{id}/verifyotp/{otp}")

**public** ResponseEntity<String> loginSecondStep(@PathVariable **int** id,@PathVariable **long** otp) **throws** OtpInvalidException, UserNotFoundException{

**try** {

String response = loginService.loginUserSecondStep(id, otp);

**return** ResponseEntity.*ok*(response);

}**catch**(OtpInvalidException e) {

**throw** **new** OtpInvalidException(e.getMessage());

}**catch**(UserNotFoundException e) {

**throw** **new** UserNotFoundException(e.getMessage());

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***BAD\_REQUEST***).body(e.getMessage());

}

}

@PutMapping("/changepassword/{username}/{newPassword}")

**public** ResponseEntity<String> changePassword(@PathVariable String username,@PathVariable String newPassword) **throws** UserNotFoundException{

**try** {

String response = loginService.changePassword(username, newPassword);

**return** ResponseEntity.*ok*(response);

}**catch**(UserNotFoundException e) {

**throw** **new** UserNotFoundException(e.getMessage());

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***BAD\_REQUEST***).body(e.getMessage());

}

}

}

@GetMapping("/getuser/{id}")

**public** ResponseEntity<User> getUserById(@PathVariable **int** id) **throws** UserNotFoundException{

**try** {

User user = loginService.getUserById(id);

**return** ResponseEntity.*ok*(user);

}**catch**(UserNotFoundException e) {

**throw** **new** UserNotFoundException(e.getMessage());

**DashboardController.java:** - The below code is related to dashboard controller where dashboard related endpoints are exposed here. I have shared code snippet of important endpoints like get products, get product details by id and get categories etc. The implementation of the respective endpoint would be in service layer and database related action would be done in repository layer. Required entities are created for dashboard changes to interact with tables in database. Custom exception handling is done to throw user defined exceptions. Configuration related to dashboard changes would be available in resources folder. The unit tests are written in src/test/java layer for functional testing and code coverage.

@RestController

@RequestMapping("/ppms/dashboard")

**public** **class** DashboardController {

@GetMapping("/getProducts")

**public** ResponseEntity<List<Products>> getAllProducts() {

**try** {

List<Products> products = dashboardService.getAllProducts();

**return** **new** ResponseEntity<>(products, HttpStatus.***OK***);

}**catch**(ResourceNotFoundException e) {

**return** ResponseEntity.*status*(HttpStatus.***NOT\_FOUND***).body(**null**);

}

}

@GetMapping("/getProductDetails")

**public** ResponseEntity<Products> getProductDetails(@RequestParam **long** productId) {

**try** {

Products product = dashboardService.getProductDetails(productId);

**return** **new** ResponseEntity<>(product, HttpStatus.***OK***);

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***NOT\_FOUND***).body(**null**);

}

}

@GetMapping("/getCategories")

**public** ResponseEntity<List<CategoryDataDTO>> getCategories() {

**try** {

List<CategoryDataDTO> categories = dashboardService.getAllCategories();

**return** **new** ResponseEntity<>(categories, HttpStatus.***OK***);

}**catch**(ResourceNotFoundException e) {

**return** ResponseEntity.*status*(HttpStatus.***NOT\_FOUND***).body(**null**);

}

}

**DashboardController.java:** - The below code is related to dashboard controller where dashboard related endpoints are exposed here. I have shared code snippet of important endpoints like get products by category , get location details by username and add Vehicle Information etc. The implementation of the respective endpoint would be in service layer and database related action would be done in repository layer. Required entities are created for dashboard changes to interact with tables in database. Custom exception handling is done to throw user defined exceptions. Configuration related to dashboard changes would be available in resources folder. The unit tests are written in src/test/java layer for functional testing and code coverage.

@GetMapping("/getProductsByCategory")

**public** ResponseEntity<List<Products>> getProductsByCategories(@RequestParam String category) {

**try** {

List<Products> products = dashboardService.getProductsByCategory(category);

**return** **new** ResponseEntity<>(products, HttpStatus.***OK***);

}**catch**(ResourceNotFoundException e) {

**return** ResponseEntity.*status*(HttpStatus.***NOT\_FOUND***).body(**null**);

}

}

@GetMapping("/getLocation")

**public** ResponseEntity<LocationDataDTO> getNearestLocation(@RequestParam String username) {

**try** {

LocationDataDTO locations = dashboardService.getNearestLocation(username);

**return** **new** ResponseEntity<>(locations, HttpStatus.***OK***);

}**catch**(ResourceNotFoundException e) {

**return** ResponseEntity.*status*(HttpStatus.***NOT\_FOUND***).body(**null**);

}

}

@PostMapping("/addVehicleInfo/{userId}")

**public** ResponseEntity<String> addVehicleInfo(@PathVariable **int** userId, @RequestBody Vehicles vehicleInfo) {

**try** {

String status = dashboardService.addVehicle(userId,vehicleInfo);

**return** **new** ResponseEntity<String>(status, HttpStatus.***OK***);

}**catch**(Exception e) {

e.printStackTrace();

**return** ResponseEntity.*status*(HttpStatus.***NOT\_FOUND***).body(**null**);

}

}

**PaymentInfoController.java:** - The below code is related to payment info controller where payment related endpoints are exposed here. I have shared code snippet of complete payment. The implementation of the respective endpoint would be in service layer and database related action would be done in repository layer. Required entities are created for payment info changes to interact with tables in database. Custom exception handling is done to throw user defined exceptions. Configuration related to payment info changes would be available in resources folder. The unit tests are written in src/test/java layer for functional testing and code coverage.

@RestController

@RequestMapping("/ppms/payment")

**public** **class** PaymentInfoController {

@GetMapping("/{userId}/success")

**public** ResponseEntity<?> successPay(@RequestParam("paymentId") String paymentId,

@RequestParam("PayerID") String payerId, @PathVariable **long** userId) {

**try** {

Payment payment = paymentInfoService.executePayment(paymentId, payerId);

**if** (payment.getState().equals("approved")) {

String currency = payment.getTransactions().get(0).getAmount().getCurrency();

String totalAmount = payment.getTransactions().get(0).getAmount().getTotal();

PaymentInfo paymentInfo = paymentInfoRepository.findById(paymentId)

.orElse(**new** PaymentInfo());

paymentInfo.setPaymentId(paymentId);

paymentInfo.setPayerId(payerId);

paymentInfo.setAmount(Double.*parseDouble*(totalAmount));

paymentInfo.setCurrency(currency);

paymentInfo.setPaymentStatus(PaymentStatus.***SUCCESS***.getStr());

paymentInfoService.addOrUpdatePayment(paymentInfo);

String redirectUrl = "http://localhost:3000/payment-success?paymentId=" + paymentId + "&status=success";

**return** ResponseEntity.*status*(HttpStatus.***FOUND***).location(URI.*create*(redirectUrl)).build();

// return ResponseEntity.ok("Payment successful. ID: " + paymentId);

} **else** {

**return** ResponseEntity.*unprocessableEntity*().body("Payment execution failed. State: " + payment.getState());

**PaymentInfoController.java:** - The below code is related to payment info controller where payment related endpoints are exposed here. I have shared code snippet of cancel payment. The implementation of the respective endpoint would be in service layer and database related action would be done in repository layer. Required entities are created for payment info changes to interact with tables in database. Custom exception handling is done to throw user defined exceptions. Configuration related to payment info changes would be available in resources folder. The unit tests are written in src/test/java layer for functional testing and code coverage.

@GetMapping("/{userId}/cancel")

**public** ResponseEntity<?> cancelPayment( @PathVariable **long** userId ) {

// Assuming paymentId is passed as a query parameter

**try** {

PaymentInfo paymentInfo = **new** PaymentInfo();

paymentInfo.setPaymentId(UUID.*randomUUID*().toString());

paymentInfo.setPayerId("EVYULUZWTXRC2");

paymentInfo.setCurrency("USD");

**if** (paymentInfo != **null**) {

paymentInfo.setPaymentStatus(PaymentStatus.***CANCELLED***.getStr());

paymentInfoService.addOrUpdatePayment(paymentInfo);

}

String redirectUrl = "http://localhost:3000/cart";

**return** ResponseEntity.*status*(HttpStatus.***FOUND***).location(URI.*create*(redirectUrl)).build();

} **catch** (Exception e) {

e.printStackTrace();

**return** ResponseEntity.*status*(HttpStatus.***INTERNAL\_SERVER\_ERROR***)

.body("An error occurred while cancelling the payment");

}

}

**CustomerController.java:** - The below code is related to customer controller where customer related endpoints are exposed here. I have shared code snippet of update personal details, add items to cart and remove items from cart. The implementation of the respective endpoint would be in service layer and database related action would be done in repository layer. Required entities are created for customer changes to interact with tables in database. Custom exception handling is done to throw user defined exceptions. Configuration related to customer changes would be available in resources folder. The unit tests are written in src/test/java layer for functional testing and code coverage.

@RestController

@RequestMapping("/ppms/customer")

**public** **class** CustomerController {

@Autowired

**private** CustomerService customerService;

@PutMapping("/update/{userId}")

**public** ResponseEntity<String> updateCustomerDetails(@PathVariable **int** userId, @RequestBody Customer customer) **throws** CustomerNotFoundException{

**try** {

String str = customerService.updateCustomer(userId,customer);

**return** **new** ResponseEntity<>(str, HttpStatus.***OK***);

}**catch**(CustomerNotFoundException e) {

**throw** **new** CustomerNotFoundException(e.getMessage());

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***BAD\_REQUEST***).body(e.getMessage());

}

}

@PostMapping("/{userId}/cart/addproduct")

**public** ResponseEntity<String> addProductToCart(@PathVariable **long** userId, @RequestBody Cart cart){

**try** {

String str = customerService.addProductToCart(cart);

**return** **new** ResponseEntity<>(str, HttpStatus.***OK***);

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***BAD\_REQUEST***).body("Error while adding item "+e.getMessage());

}

}

@DeleteMapping("/{userId}/cart/remove/{productId}")

**public** ResponseEntity<String> removeProductFromCart(@PathVariable **long** userId, @PathVariable **long** productId){

**try** {

String str = customerService.removeProductFromCart(productId, userId);

**return** **new** ResponseEntity<>(str, HttpStatus.***OK***);

}**catch**(Exception e) {

**return** ResponseEntity.*status*(HttpStatus.***BAD\_REQUEST***).body("Error while removing item "+e.getMessage());

}

}

**For Front end:**

**LoginComponent:** The below code is related to login component where login related screens would be designed. Service integrations would be done to call back end API’s and the response would be populated on the screen. Routes to the login pages would be configured in main app folder. CSS related changes would be done using inline CSS, external CSS, styled components, and bootstrap. Global state storage is used to handle and expose data to all components. Different hooks are used for state management.

export default function LoginComponent({ lightTopLine }) {

const navigate = useNavigate();

const [hover, setHover] = useState(false);

const [state, dispatch] = useContext(GlobalStateContext);

const [username, setUsername] = useState('');

const [password, setPassword] = useState('');

const [otp, setOtp] = useState('')

const [showOtpModal, setShowOtpModal] = useState(false);

const onHover = () => {

setHover(!hover);

};

const handleLogin = async (event) => {

event.preventDefault()

const response = await LoginServices.login(username, password).then(resp => {

if (resp.data.message === 'Enter OTP generated' && resp.status === 200) {

setShowOtpModal(true);

dispatch({ type: 'SET\_USER\_DATA', payload: resp.data });

}

}).catch(err => {

console.log(err)

})

}

const handleOtpSubmit = async (event) => {

event.preventDefault()

// Logic to handle OTP submission

console.log("sssss", state.userData.userId)

const otpResponse = await LoginServices.verifyOtp(state.userData.userId, otp).then(

resp => {

setShowOtpModal(false);

navigate("/dashboard");

}

).catch(err => {

console.log(err)

})

**Dashboard:** The below code is related to dashboard component where dashboard related screens would be designed. The dashboard contains dashboard navbar, products and categories. Service integrations would be done to call back end API’s and the response would be populated on the screen. Routes to the dashboard pages would be configured in main app folder. CSS related changes would be done using inline CSS, external CSS, styled components, and bootstrap. Global state storage is used to handle and expose data to all components. Different hooks are used for state management.

const Dashboard = () => {

return (

<>

<DashboardNavbar>

</DashboardNavbar>

<Products />

<Categories/>

</>

);

};

function Products() {

const navigate = useNavigate();

const [data, setData] = useState([]);

const [loading, setLoading] = useState(true)

const handleProductClick = (event,product) => {

console.log(event)

navigate(`/productdetails/${product.productId}`, { state: { product } });

};

const getData = () => {

DashboardServices.getAllProducts().then(response => {

console.log(response.data)

const modifiedData = response.data.map(item => ({

productId: item.productId,

productName: item.productName,

productImage: generateImage(item.productImage),

productDescription: item.productDescription,

productPrice: item.productPrice

}));

setData(modifiedData);

setLoading(e => !e)

}).catch(error => {

console.log(error)

})

}

export default Dashboard;

**ProductsByCategory:** The below code is related to products by category component where product details related screens would be designed. Service integrations would be done to call back end API’s and the response would be populated on the screen. Routes to the products by category pages would be configured in main app folder. CSS related changes would be done using inline CSS, external CSS, styled components, and bootstrap. Global state storage is used to handle and expose data to all components. Different hooks are used for state management.

const ProductsByCategory = () => {

const [state, dispatch] = useContext(GlobalStateContext);

const location = useLocation();

const [loading, setLoading] = useState(true)

const [products, setProducts] = useState([])

const categoryName = location.state.categoryName;

const [hover, setHover] = useState(false);

const onHover = () => {

setHover(!hover);

};

const getData = () => {

DashboardServices.getProductsByCategoryName(categoryName).then(response => {

console.log(response.data)

const modifiedData = response.data.map(item => ({

productId: item.productId,

productName: item.productName,

productImage: generateImage(item.productImage),

productDescription: item.productDescription,

productPrice: item.productPrice

}));

setProducts(modifiedData)

setLoading(e => !e)

}).catch(error => {

console.log(error)

})

}

const handleCart=(event,item)=>{

const existingProduct = state.cart.find(cartItem => cartItem.id === item.productId);

if (existingProduct) {

dispatch({

type: 'INCREASE\_QUANTITY',

payload: item.productId

});

**AddVehicleDetails:** The below code is related to add vehicle details component where vehicle registration would be designed. Service integrations would be done to call back end API’s and the response would be populated on the screen. Routes to the add vehicle details pages would be configured in main app folder. CSS related changes would be done using inline CSS, external CSS, styled components, and bootstrap. Global state storage is used to handle and expose data to all components. Different hooks are used for state management.

const AddVehicleDetails = ({lightTopLine}) => {

const [hover, setHover] = useState(false);

const navigate = useNavigate();

const [loading, setLoading] = useState(false)

const [state] = useContext(GlobalStateContext);

const [vehicle, setVehicle] = useState({

vehicleYear: '',

licenseplate: '',

vehicleMaker:'',

vehicleModel:'',

vehicleIdNumber:'',

mileage:'',

vehicleType:'',

color:'',

vehicleUsage:''

});

const onHover = () => {

setHover(!hover);

};

const handleChange = (e) => {

setVehicle({ ...vehicle, [e.target.name]: e.target.value });

};

const handleSubmit = (e) => {

e.preventDefault();

console.log(vehicle);

DashboardServices.addVehicleDetails(state.userData.userId, vehicle).then(response => {

console.log(response.data)

setLoading(e => !e)

navigate('/dashboard')

}).catch(error => {

console.log(error)

})

};

**CartComponent:** The below code is related to cart component where cart screens would be designed. Service integrations would be done to call back end API’s and the response would be populated on the screen. Routes to the cart pages would be configured in main app folder. CSS related changes would be done using inline CSS, external CSS, styled components, and bootstrap. Global state storage is used to handle and expose data to all components. Different hooks are used for state management.

const CartComponent = () => {

const [state, dispatch] = useContext(GlobalStateContext);

const [cart, setCart] = useState({

totalAmount :''

});

console.log(state.cart)

const increaseQuantity = (id) => {

dispatch({ type: 'INCREASE\_QUANTITY', payload: id });

};

const decreaseQuantity = (id) => {

dispatch({ type: 'DECREASE\_QUANTITY', payload: id });

};

const removeItem = (id) => {

dispatch({ type: 'REMOVE\_FROM\_CART', payload: id });

};

const calculateTotal = () => {

return state.cart.reduce((acc, item) => acc + item.productPrice \* item.quantity, 0);

};

const handlePayment = (event) => {

const totalAmount=calculateTotal().toFixed(2)

event.preventDefault()

setCart(prevCart => ({

...prevCart,

totalAmount: totalAmount

}));

const response = PaymentServices.createPayment(state.userData.userId, cart).then(resp => {

console.log(response)

if (resp.status === 200) {

window.open(resp.data, '\_blank');

}

}).catch(err => {

console.log(err)

})

};

**UpdatePersonalDetails:** The below code is related to update personal details component where personal details registration would be designed. Service integrations would be done to call back end API’s and the response would be populated on the screen. Routes to the update personal details pages would be configured in main app folder. CSS related changes would be done using inline CSS, external CSS, styled components, and bootstrap. Global state storage is used to handle and expose data to all components. Different hooks are used for state management.

const UpdatePersonalDetails = ({ lightTopLine }) => {

const [hover, setHover] = useState(false);

const navigate = useNavigate();

const [loading, setLoading] = useState(false)

const [state] = useContext(GlobalStateContext);

const [customer, setCustomer] = useState({

fullName: '',

emailId: '',

mobileNumber: '',

zipcode: '',

});

const onHover = () => {

setHover(!hover);

};

const renderInputField = (key, value) => {

switch (key) {

case 'mobileNumber':

return (

<input

style={{

borderRadius: "6px",

textAlign: "center",

padding: "0.7vh",

marginLeft: "1vh",

}}

type="number"

name={key}

value={value}

onChange={handleChange}

/>

);

const handleSubmit = (e) => {

e.preventDefault();

console.log(customer);

CustomerServices.updatePersonalDetails(state.userData.userId, customer).then(response => {

console.log(response.data)

setLoading(e => !e)

navigate('/dashboard')