

Swoplet

Milestone 4: Next Beta Launch, QA and Usability Testing and Final Commitment for Product Features (P1 list)

Team 6

Team Lead Email: muhammad-hamza.butt@informatik.hs-fulda.de

Team Member	Role
Hamza Butt	Team Lead
Karan Patel	Git master
Pooja Puthu Vayal	Backend Lead
Akhil Sajan	Frontend Lead
Hammad	Frontend Developer
Hasara Koralege	Backend Developer

History

Date	Version
11/07/2025	First Draft

1. Product summary

Purchasing, selling, and exchanging products within the campus community can provide challenges for university staff, faculty, and students. Their particular needs are usually not met by public platforms, raising questions about security, pricing fairness, and item authenticity.

Swoplet is a safe online platform created just for the Fulda University community to close this gap. It guarantees that the platform can only be accessed and used by confirmed university members to safely purchase, sell, or exchange goods and services. With a university-verified user base, moderator oversight for increased security, and an AI-powered pricing assistant that assists users in estimating fair market prices prior to transactions, Swoplet provides a smooth user experience that is suited to the demands of campus commerce. This encourages transparency and empowers users to make well-informed decisions.

Final Committed Major P1 Functions

- Users must register with their official university email address.
- Users can browse available listings on the platform.
- Users can search for specific listings using keywords.
- Users can filter and sort listings by category, price, or time posted.
- Buyers can securely message sellers through an internal messaging system.
- Buyers can save items to a wishlist for future reference.
- Sellers can create new listings with item details like title, price, and description.
- Sellers can upload images of their product.
- Sellers can respond to buyer inquiries using the internal messaging system.
- Sellers have a dashboard to manage listings (edit, delete, mark sold, etc.).
- Sellers can temporarily hide a listing from public view.
- Sellers must categorize listings properly and add relevant tags.
- Admin must approve listings before they are visible to others.
- Admin can moderate the platform by removing content or banning users.

Unique Selling Point:

Includes an AI-based pricing suggestion feature that recommends fair market prices based on current listing trends, aiding both buyers and sellers in making informed decisions.

Deployment URL:

[Swoplet](#)

2. Usability test plan

2.1. Test objectives

- **Effectiveness:**
Can sellers complete the full process of posting a listing with all required fields, images, and metadata without any help?
- **Efficiency:**
How long does it take users to go from the **dashboard** to a **successfully published listing**?
Where do they get stuck?
- **Satisfaction:**
How do users rate their overall experience in terms of usability, communication, reliability, and performance?

2.2. Test Background & Setup

System Under Test:

[Swoplet Live Site](#)

- **Test Environment:**
 - Devices: Laptop, desktop, mobile phones
 - Browsers: Chrome (latest), Edge (latest), Safari (mobile)
 - Network: Stable high-speed Internet
- **Test Participants:**
 - 5–10 real users or representative testers (preferably unfamiliar with the platform)

2.3. Test Roles

Role	Responsibility
Facilitator	Gives instructions, observes behavior, avoids interfering
Note-taker	Logs issues, confusion points, time metrics
User/Test Subject	Completes tasks while thinking aloud

2.4. Test Tasks & Instructions

1. Login

- a. Go to the Swoplet homepage and log in using the provided seller's test credentials or create a new user.

2. Navigate

- a. From the dashboard, locate and click the **“Post New Item”** button.

3. Create Listing

- a. Fill in product listing details:
 - i. Realistic **title, description**
 - ii. Select appropriate **category**
 - iii. Add **price, availability**
 - iv. Upload **minimum 2 images**
 - v. Add relevant
- b. Click **“Create Listing”** to complete.

4. Verify Listing

- a. Go to **“My Listings”** and confirm that the new item appears correctly.

5. Post-Task

- o Complete the **Satisfaction Questionnaire** (see 2.1.6).

2.5. Metrics & Data Collection

Metric	Description
Success Rate	% of users who publish a listing without errors
Time on Task	Time taken from dashboard to published listing
Error Frequency	Upload failures, field errors, navigation misclicks
User Comments	Verbal or written feedback

Observations	Points of hesitation, repeated attempts, UI confusion
---------------------	---

2.6. Post-Test Satisfaction Questionnaire

All questions use a 1–5 Likert scale (1 = Strongly Disagree, 5 = Strongly Agree)

Ease of Use

- Swoplet is easy to navigate and use.
- Creating and managing product listings is straightforward.
- Finding products, I want to buy is efficient.

Communication

- The chat system makes communication with other users easy.
- I feel comfortable and safe when interacting with other users.

Trust & Reliability

- The seller rating system helps me make informed decisions.
- The platform feels secure and trustworthy.

Performance

- The website loads quickly and works smoothly.
- I rarely encounter errors or technical issues.

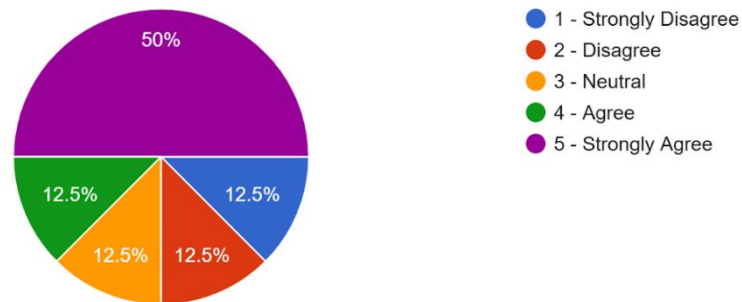
Overall Satisfaction

- I am satisfied with my overall Swoplet experience.
- I would recommend Swoplet to others.
- I plan to continue using Swoplet in the future.

2.7.Feedback Analysis

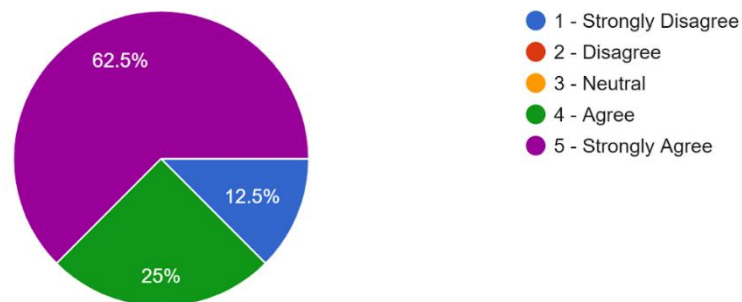
Swoplet is easy to navigate and use.

8 responses



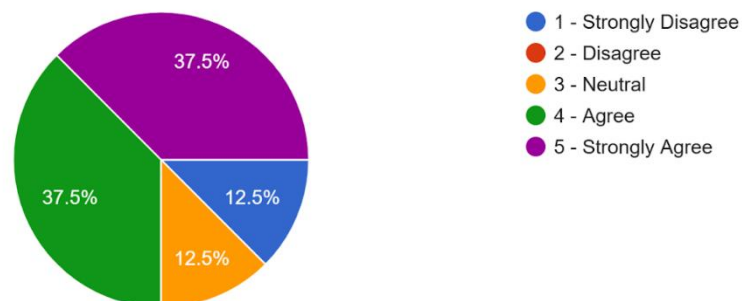
Creating and managing product listings is straightforward.

8 responses



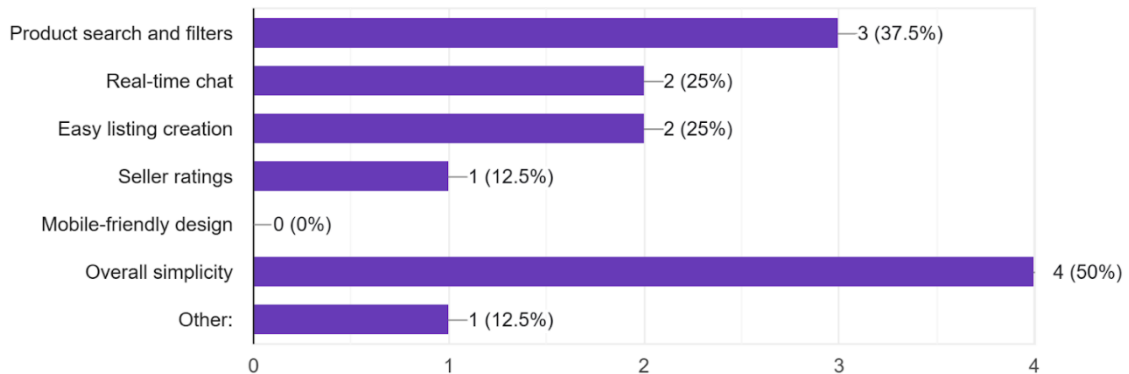
I am satisfied with my overall Swoplet experience.

8 responses



What's your favorite feature of Swoplet?

8 responses



3. QA test plan

3.1. Test Objectives

The primary objective of this QA test plan is to ensure the functionality and reliability of the "Create Product Listing" feature. To ensure that users can successfully create a new product listing with required fields (title, description, price, tags, images, etc.), and that form validations, file uploads, and API responses work as expected across different browsers.

3.2. HW and SW setup

Component	Details
OS	Windows 10 / macOS Ventura
Browsers	Google Chrome (v124+) and Microsoft Edge (v124+)
Device	Laptop/Desktop
URL	swoplet-zler.onrender.com
Backend API	https://swoplet-zler.onrender.com/api/
Database	swoplet-db.cx4q28o46i78.eu-central-1.rds.amazonaws.com (Amazon RDS)

3.3.Feature to be tested

- Product creation form on the "Create Listing" page
- Field validations (required, format)
- Media file upload and preview
- API submission and response
- UI feedback after successful listing

3.4.QA Test plan

Test #	Test Title	Test Description	Test Input	Expected Output	Chrome Result	Edge Result
1	Create Listing – Success	Submit all required fields to create listing successfully	Title: "Hair Dryer" Description: "HH Simonsen Compact Dryer, black. Immediately available, delivery time: 1-3 days" Price: \$99 Category: Appliances Condition: Used-Good Tags: "hair dryer, used" Image: JPEG	After creating the listing, a notification appears in the UI stating that the product was successfully created. The user stays on the same page, and the form is reset.	PASS	PASS
2	Create Listing Success & My Listings	Create listing successfully and check user's listing page	Same input as Test 1	The new product listing appears on the user's My Listings page under the Pending category.	PASS	PASS
3	Create Listing Success & Admin Dashboard	Create listing successfully and check admin dashboard	Same input as Test 1	The new product listing appears on the Admin Dashboard page under the Pending list.	PASS	PASS
4	Missing Required Fields	Submit form without title or price	Title: "" Price: "" Other fields filled	Validation error messages are displayed for	PASS	PASS

Test #	Test Title	Test Description	Test Input	Expected Output	Chrome Result	Edge Result
				required fields; form is not submitted.		
5	Image Upload Validation	Upload a file with an unsupported format	Upload: document.pdf	Error message displayed: "Only image files are allowed."	PASS	PASS
6	Large Image Upload	Try uploading an image larger than 5MB	Upload: image_6mb.jpg	Error displayed: "File size exceeds limit," or upload fails gracefully.	PASS	PASS
7	Create Without Category	Submit listing without selecting a category	Leave category field empty	Validation error displayed: "Please select a category."	PASS	PASS

4. Code Review

4.1. Coding Style

- Consistent use of whitespaces (for example, nested code should be tabbed in)
- Names for functions, classes, variables, etc. need to clearly describe what they are used for/what they do
- Naming is done in camelCase (example: "getListingData()")
- DRY (Don't Repeat Yourself) principle: No duplicate code; if the same code is needed multiple times, it will be moved into a function-call
- KISS (Keep It Simple, Stupid) principle: Code is only as complex as it needs to be
- Code that isn't straightforward has comments that describe what it does
- Proper Error Handling is done (example: try-catch)
- A clean file order is used (example: controller calls service calls repository)

4.2. Review: AdminDashboard.jsx

Code writer: Akhil Sajan

Reviewer: Pooja Pv

Date: 27.06.2025

File to be reviewed: main/UI/src/Pages/AdminDashboard/AdminDashboard.jsx

Overview

The AdminDashboard component is part of an admin panel interface allowing administrators to manage product listings and monitor reported users. It fetches listings based on their approval status (pending, approved, rejected) and allows admins to approve/reject pending listings. It also provides a tab to view reported users.

Strengths and Positive Aspects

- **Modular & Clean Structure:** The code is split into logical sections: state management, side effects (useEffect), and handlers.
- **Reusability with useApiRequest Hook:** API calls are abstracted into a reusable custom hook, improving testability and reducing repetition.
- **Role-Based Access Control:** Includes a check to restrict access based on admin email.
- **Responsive Design:** Utilizes Tailwind CSS for a responsive, visually appealing layout.
- **Clear UI Logic:** Conditionals based on activeTab make the user interface intuitive and manageable.

Areas of Improvement

1. The same fetchProducts function is called multiple times with only the status value changing. This violates DRY (Don't Repeat Yourself) principles and can be simplified using a mapping object.

```
33     useEffect(() => {
34         if (activeTab === "pending") {
35             fetchProducts({
36                 url: `${BASE_URL}products/search`,
37                 method: "POST",
38                 body: { status: 0 },
39                 headers: { "Content-Type": "application/json" },
40             });
41         } else if (activeTab === "approved") {
42             fetchProducts({
43                 url: `${BASE_URL}products/search`,
44                 method: "POST",
45                 body: { status: 1 },
46                 headers: { "Content-Type": "application/json" },
47             });
48         } else if (activeTab === "rejected") {
49             fetchProducts({
50                 url: `${BASE_URL}products/search`,
51                 method: "POST",
52                 body: { status: 2 },
53                 headers: { "Content-Type": "application/json" },
```

2. This product rendering block is copied three times. Consider creating a small `renderProducts(products, status)` function or a reusable component to avoid repeating the same JSX. It's easier to maintain if the design changes later.

```
293         {productsData.products
294           .filter((product) => product.status === 2)
295           .map((product) => (
296             <AdminProductCard
297               key={product.id}
298               title={product.title}
299               image={
300                 product.media_link ||
301                 product.image_url ||
302                 "https://via.placeholder.com/150"
303             )
304           )
305         }
```

3. Optional improvement: Consider putting "admin@hs-fulda.de" into a config file or constant, so it's easier to maintain.
4. You can remove unused items like `searchQuery`, the `<ProductCard />` at the end, and any other variables not being used to keep the code clean. Also, adding a few short comments above key functions like `handleApprove` and `handleReject` would improve readability.

4.3. Review: product listing controller

- **Code writer:** Pooja
- **Reviewer:** Akhil Sajan
- **Date:** 27.06.2025
- **File to be reviewed:** Backend/controllers/productlisting.controller.js

Overview

This controller manages core product listing operations:

- Fetching listings by status (approved, by ID, by user, etc.)
- Creating, updating, and soft-deleting listings
- Updating product status (including marking products as sold)
- Enriching listings with related data (eg, seller review stats, media files, categories)

It makes good use of Sequelize's ORM features and keeps business logic mostly inside controller functions.

Strengths and Positive Aspects

- **Separation of concerns:** Controller logic is organized cleanly, and data access/manipulation is handled by Sequelize models.
- **Error handling:** Each function is wrapped in try/catch blocks and provides clear, meaningful error messages to the client.
- **Reusability:** The helper function `getSellerReviewStats` abstracts seller rating calculation, preventing repeated aggregation logic across endpoints.
- **Soft deletion:** Uses an `is_deleted` flag to mark records instead of permanently deleting them, which supports safer data recovery.
- **Data enrichment:** Listing endpoints include related entities like media files, categories, conditions, creators, and seller review stats, improving the quality of response data.
- **Bulk operations:** Efficiently handles media replacement with `bulkCreate` and cleans up dependent data (like wishlists) during updates.
- **Clear status management:** Implements careful status tracking by resetting status to pending on updates and preserving the previous status for historical context.

Areas of Improvement

- **Code repetition:** Many `findAll` and `findByPk` calls repeat similar include clauses. Extract common includes into a constant (eg, `const PRODUCT_INCLUDES = [...]`) to DRY up the code.

```
const listings = await productlisting.findAll({
  attributes: [
    'id', 'title', 'description', 'price', 'tags', 'location', 'status', 'created_at', 'updated_at'
  ],
  where: {
    status: APPROVAL_STATUS.APPROVED,
    is_deleted: DELETE_STATUS.ACTIVE
  },
  include: [
    { model: mediafile, as: 'mediafiles', attributes: ['file_path'], where: { is_approved: true }, required: true },
    { model: productcategory, as: 'category' },
    { model: productcondition, as: 'condition' },
    { model: user, as: 'creator' }
  ]
});
```

- **Hardcoded status values:** Status values like 3(sold) and 1(approved) are used directly; replace them with named constants from `APPROVAL_STATUS` to improve readability and reduce errors.

```
await product.update({
  previous_status: product.status,
  status: 3, // 3 = sold
  updated_at: new Date().toISOString()
});
```

4.4.Review: ProductDetailPage.jsx

Reviewer: Hasara Koralege

Code writer: Hammad Asif

Date: 27.06.2025

File to be reviewed: UI\src\Pages\ProductDetail\ProductDetailPage.jsx

Overview


The ProductDetailPage.jsx file defines the product detail page, which displays comprehensive information about a selected product. It fetches product data from the backend using the product ID from the URL, and renders details such as images, description, seller information, and reviews. The page also allows users to add/remove the product from their favourites and contact the seller, with appropriate UI feedback and backend calls. Tabs are used to organize product information, and toast notifications provide user feedback for actions.

Strengths and Positive Aspects

1. The code is modular and leverages React hooks effectively for state and side effects.
2. API calls are abstracted using a custom hook (useApiRequest), improving reusability and separation of concerns.
3. The UI is responsive and well-structured, with clear separation between product details, images, and actions.

Areas of Improvement

1. Remove unwanted imports to keep code clean. You import SearchBar, but don't use it anywhere.



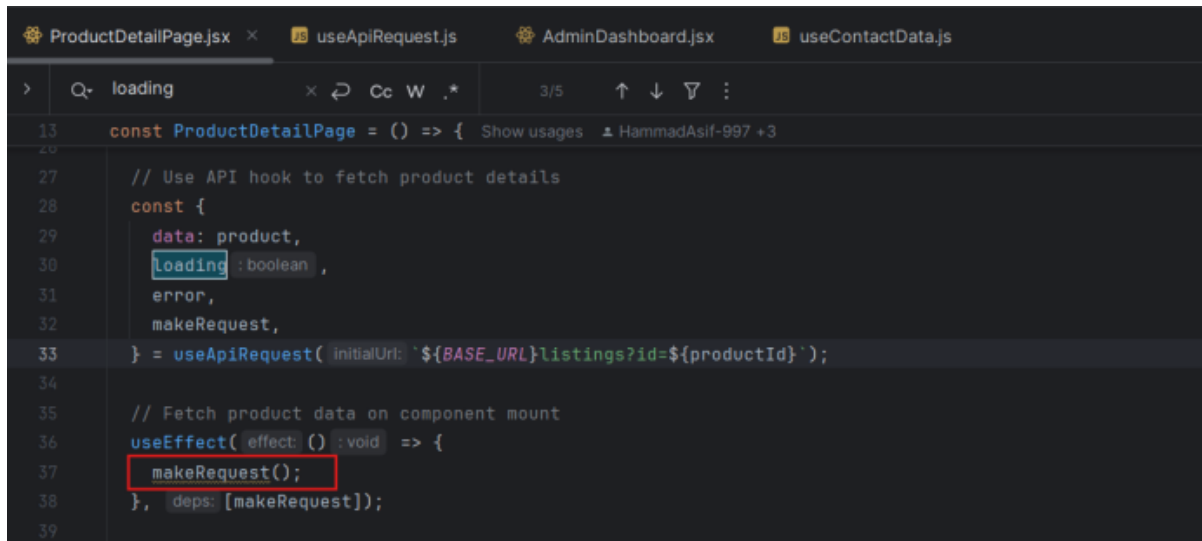
```
ProductDetailPage.jsx x
1  import { ChevronLeft, ChevronRight } from "lucide-react";
2  import SearchBar from "../../components/Common/SearchBar";
3  import React, { useState, useEffect } from "react";
4  import { useApiRequest } from "../../hooks/useApiRequest";
5  import { BASE_URL, userId } from "../../constants/config";
6  import LoadingState from "../../components/Common/LoadingState";
7  import ErrorState from "../../components/Common/ErrorState";
```

2. Avoid native alert() in modern UIs. alert() blocks the main thread, interrupts the user experience, and looks outdated. Instead add userfriendly non-blocking notification of error in configured position.

```
ProductDetailPage.jsx ×
13  const ProductDetailPage = () => { Show usages ⚡ HammadAsif-997 +3
164  const handleContactSeller = async () : Promise<void> => { Show usages ⚡ HammadAsif-997
165      // If not logged in, redirect to login
166      if (!currentUserId) {
167          navigate("/login");
168          return;
169      }
170
171      // Don't allow contacting yourself
172      if (parseInt(currentUserId) === product.created_by_id) {
173          alert("You cannot contact yourself!");
174          return;
175      }
176
177      setIsContactingSeller( value: true);
```

```
ProductDetailPage.jsx ×
13  const ProductDetailPage = () => { Show usages ⚡ HammadAsif-997 +3
164  const handleContactSeller = async () : Promise<void> => { Show usages ⚡ HammadAsif-997
186
187      // Store information about the chat we want to open
188      localStorage.setItem('activeChatInfo', JSON.stringify( value: {
189          sellerId: product.created_by_id,
190          productId: product.id,
191          sellerName: product.creator.username,
192          productTitle: product.title
193      }));
194
195      // Navigate to chat page regardless of whether chat existed or was newly created
196      navigate("/messages");
197      } catch (error) {
198          console.error("Error contacting seller:", error);
199          alert("Failed to contact seller. Please try again.");
200      } finally {
201          setIsContactingSeller( value: false);
202      }
203  };
204
```

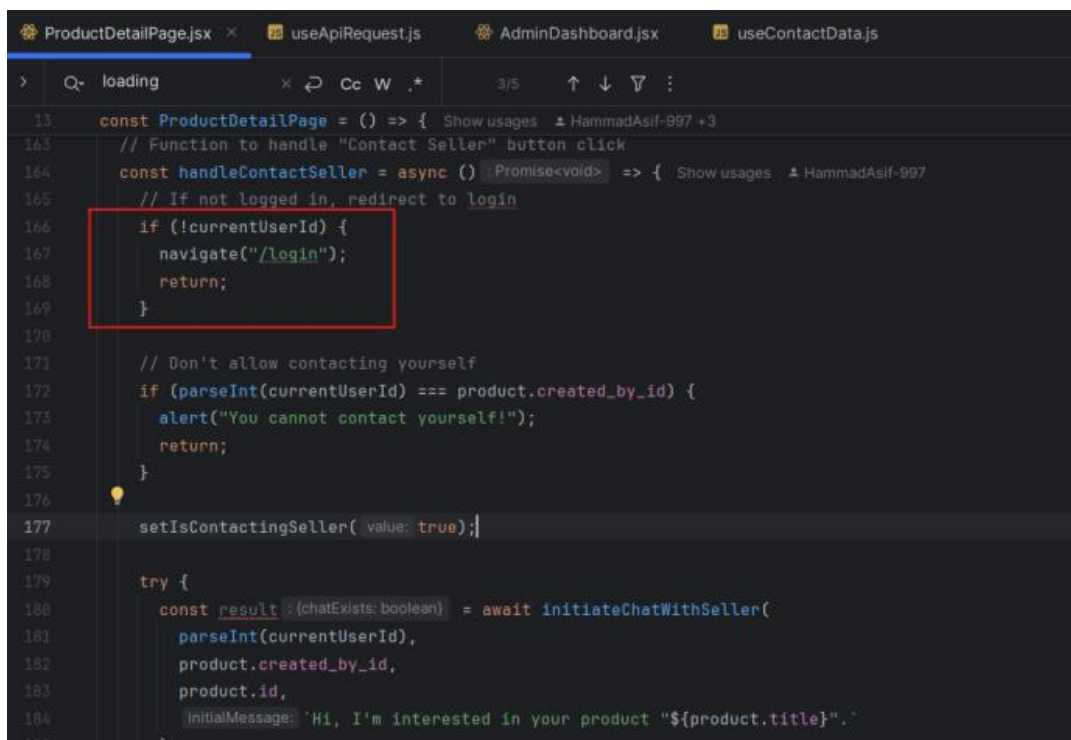
3. I recommend explicitly passing the url as an override when calling makeRequest. While useApiRequest takes initial Url on initialization, it won't update initialUrl if productId changes later without remounting the component. By explicitly providing the url to makeRequest(), we ensure the latest productId is always used for the API request, even if the component stays mounted. This makes the data fetching more robust if productId can change dynamically



The screenshot shows a VS Code editor with the file `ProductDetailPage.jsx` open. The code defines a `ProductDetailPage` component that uses the `useApiRequest` hook. The hook is called with `initialUrl: `${BASE_URL}listings?id=${productId}``. Inside the component, a `useEffect` hook is used to call `makeRequest()` when the component mounts. The `loading` state is highlighted in blue, and `makeRequest()` is highlighted in red.

```
13 const ProductDetailPage = () => { Show usages HammadAsif-997 +3
20
27 // Use API hook to fetch product details
28 const {
29   data: product,
30   loading: boolean,
31   error,
32   makeRequest,
33 } = useApiRequest( initialUrl: `${BASE_URL}listings?id=${productId}` );
34
35 // Fetch product data on component mount
36 useEffect( effect: () :void => {
37   makeRequest();
38 }, deps: [makeRequest]);
39
```

4. Extract a reusable function for the auth check + redirect and define a helper function inside your component or in a utils file.



The screenshot shows a VS Code editor with the file `ProductDetailPage.jsx` open. The code defines a `handleContactSeller` function that checks if the user is logged in. If not, it redirects to the login page. The `if (!currentUserId) {` block is highlighted in red. The function also checks if the user is the creator of the product and sets the `isContactingSeller` state to true. The `try {` block is highlighted in blue.

```
13 const ProductDetailPage = () => { Show usages HammadAsif-997 +3
163 // Function to handle "Contact Seller" button click
164 const handleContactSeller = async () :Promise<void> => { Show usages HammadAsif-997
165   // If not logged in, redirect to login
166   if (!currentUserId) {
167     navigate("/login");
168     return;
169   }
170
171   // Don't allow contacting yourself
172   if (parseInt(currentUserId) === product.created_by_id) {
173     alert("You cannot contact yourself!");
174     return;
175   }
176
177   setIsContactingSeller( value: true );
178
179   try {
180     const result : {chatExists: boolean} = await initiateChatWithSeller(
181       parseInt(currentUserId),
182       product.created_by_id,
183       product.id,
184       {initialMessage: `Hi, I'm interested in your product "${product.title}"`},
185     );
186   }
187 }
```

```
13 const ProductDetailPage = () => { Show usages HammadAsif-997 +3
112 // Function to handle "Add to Favourite" or "Remove from Favourite" button click
113 const handleFavouriteToggle = async () : Promise<void> => { Show usages HammadAsif-99
114 // If not logged in, redirect to login
115 if (!currentUserId) {
116   navigate("/login");
117   return;
118 }
119
120 setIsAddingToFavourite( value: true);
121
122 try {
123   if (isProductInFavourite) {
124     // Remove from favourites
125     const response : Response = await fetch(
126       input: `${BASE_URL}favourites?User_id=${currentUserId}&product_id=${product
127       init: { method: "DELETE" }
128     );
```

4.5. Review: sellerreview.controller.js

- **Code writer:** Hasara Koralege
- **Reviewer:** Hammad Asif
- **Date:** 27.06.2025
- **File to be reviewed:** Backend/controllers/sellerreview.controller.js

Overview

This file [seller-review.controller.js](#) provides two main controller methods:

- **getPendingSellerReviews:** Fetches sellers who a given buyer is yet to review, along with product and seller data.
- **addSellerReview:** Saves a new review for a seller, marks the corresponding seller_to_review entry as reviewed, and stores the rating and optional comment.

The module ensures buyers are reminded to review sellers after transactions and supports a transparent review system with average rating tracking.

Strengths and Positive Aspects

- Code is clean, readable, and logically structured.
- Average rating and total count are dynamically calculated with helper function `getSellerReviewStats()`.
- Effective use of Sequelize's include and conditional querying.

- Error handling is properly implemented using try/catch.
- Clear response messages and appropriate HTTP status codes are used.
- Makes good use of optional chaining (e.g., `entry.product?.mediafiles[0]?.file_path`).
- Handles nullable values like `review_comment` gracefully.

Areas for Improvement

- **Performance Optimization:** The use of `Promise.all()` for each seller review entry might create N additional DB calls for `getSellerReviewStats()`. This could be optimized with a batch query or caching mechanism.
- **Commenting:** Some logic blocks (especially within `map`) would benefit from brief comments.
- **Validation Layer:** Input validation should ideally be extracted into middleware (e.g., using `express-validator`) for cleaner controller code.
- **File Organization:** Moving `getSellerReviewStats()` to a service layer would make the controller slimmer and more testable.
- **Typo Risk:** Consistency in object keys could be improved (`reviewee_id` vs. `reviewer_id` — confirm naming aligns across models).

Specific Code Comments

Figure	LINE	COMMENT
1	5-14	Good use of helper function to centralize average rating logic. Could be moved to a service.
2	23-25	Input validation could be moved to middleware.
3	29-41	Efficient data fetching using Sequelize <code>include</code> with aliasing. Good use of <code>required: false</code> .
4	43-54	<code>Promise.all()</code> with nested DB calls could impact performance for large datasets.
5	92	Considers optional <code>review_comment</code> . Nice defensive programming.

```

3 // Helper function to get seller's total reviews and average rating
4 async function getSellerReviewStats(seller_id) {
5   const reviews = await sellerReview.findAll({
6     include: [{
7       model: sellerToReview,
8       as: 'toReview',
9       where: { seller_id },
10      attributes: []
11    }],
12    attributes: ['review_value']
13  });
14  const total = reviews.length;
15  const sum = reviews.reduce((acc, r) => acc + (r.review_value || 0), 0);
16  const avg = total > 0 ? sum / total : 0;
17  return { total, avg };
18 }

```

Figure 1: seller-review.controller.js Code 1

```

20 // GET /seller-to-review?buyer_id=xxx
21 exports.getPendingSellerReviews = async (req, res) => {
22   try {
23     const { buyer_id } = req.query;
24     if (!buyer_id) {
25       return res.error('buyer_id is required', 400);
26     }
27   }

```

Figure 2: seller-review.controller.js Code 2

```

28 // Find all seller_to_review entries for this buyer where is_reviewed is false
29 const entries = await sellerToReview.findAll({
30   where: {
31     buyer_id,
32     is_reviewed: false
33   },
34   include: [
35     { model: user, as: 'seller', attributes: ['id', 'username'] },
36     { model: productlisting, as: 'product', attributes: ['id', 'title', 'location'],
37       include: [
38         { model: mediafile, as: 'mediafiles', attributes: ['file_path'], required: false, limit: 1 }
39       ]
40     }
41   ]
42 });

```

Figure 3: seller-review.controller.js Code 3

```

43
44 // For each entry, get seller stats
45 const result = await Promise.all(entries.map(async entry => {
46   const stats = await getSellerReviewStats(entry.seller_id);
47   return {
48     id: entry.id,
49     seller_id: entry.seller_id,
50     seller_name: entry.seller?.username,
51     product_id: entry.product_id,
52     product_name: entry.product?.title,
53     location: entry.product?.location,
54     purchase_date: entry.purchase_date,
55     media_file: entry.product?.mediafiles && entry.product.mediafiles.length > 0 ? entry.product.mediafiles[0].file_path : null,
56     seller_average_rating_value: stats.avg,
57     seller_total_rating: stats.total
58   };
59 }));
60

```

Figure 4: seller-review.controller.js Code 4

```

86 // Create seller_review record
87 const review = await sellerReview.create({
88   reviewee_id,
89   reviewer_id,
90   to_review_id,
91   review_value,
92   review_comment: review_comment || null
93 });

```

Figure 5: seller-review.controller.js Code 5

4.6. Review - ChatHeader.jsx

- **Code Reviewer:** Hamza Butt
- **Code Owner:** Karan Patel
- **Date:** 27.06.2025
- **File to be reviewed:** UI/src/components/Chat/ChatHeader.jsx

Suggestions:

Extract this array into a shared constants file. Keeping “commonReasons” here makes it harder to update or reuse.

```

// Common report reasons for quick selection
const commonReasons = [
  'Inappropriate behavior',
  'Spam or scam',
  'Harassment',
  'Fake profile',
  'Inappropriate content'
];

```

Move localStorage helpers (getReportedUsers, addReportedUser, hasReportedUser) into a separate file (useReportUtils). This will be helpful to maintain single responsibility.

```
// Utility functions for managing reported users (consolidated)
const getReportedUsers = (currentUserId) => {
  try {
    const reportedUsers = localStorage.getItem(`reportedUsers_${currentUserId}`);
    return reportedUsers ? JSON.parse(reportedUsers) : [];
  } catch (error) {
    console.error('Error getting reported users:', error);
    return [];
  }
};

const addReportedUser = (currentUserId, reportedUserId) => {
  try {
    const reportedUsers = getReportedUsers(currentUserId);
    if (!reportedUsers.includes(reportedUserId)) {
      reportedUsers.push(reportedUserId);
      localStorage.setItem(`reportedUsers_${currentUserId}`, JSON.stringify(reportedUsers));
    }
  } catch (error) {
    console.error('Error adding reported user:', error);
  }
};

const hasReportedUser = (currentUserId, userIdToCheck) => {
  const reportedUsers = getReportedUsers(currentUserId);
  return reportedUsers.includes(userIdToCheck);
};
```

This is pure validation logic. pull it out into its own file like chatHeaderValidation.js. This helps to keep the component UI focused.

```
const validateReportReason = (reason) => {
  const trimmedReason = reason?.trim() || '';

  if (trimmedReason.length < 5) {
    return {
      isValid: false,
      message: 'Reason must be at least 5 characters long'
    };
  }

  if (trimmedReason.length > 500) {
    return {
      isValid: false,
      message: 'Reason must be less than 500 characters'
    };
  }

  return {
    isValid: true,
    message: 'Valid reason'
  };
};
```

Replace the number “3” with an enum or named constant, e.g.

```
const PRODUCT_STATUS = { AVAILABLE: 1, REJECTED: 2, SOLD: 3 };
```

```
const isProductSold = productStatus === 3;
```

All of the text can be moved to a constant file. It will make it more readable and easy to modify.

```

const handleConfirmSold = async () => {
  if (!productId || !contact.id) {
    setSoldMessage('Error: Missing product or buyer information.');
```

`return;`

```
  }

  setIsMarkingSold(true);
  setSoldMessage('');

  try {
    const response = await markProductAsSoldAPI(productId, contact.id);

    if (response.success) {
      setSoldMessage('Product marked as sold successfully! This will help other users know it\'s no longer av
      setTimeout(() => {
        setIsSoldDialogOpen(false);
        setSoldMessage('');
        window.location.reload();
      }, 2000);
    } else {
      setSoldMessage(response.error || 'Failed to mark product as sold. Please try again.');
```

`}`

```
  } catch (error) {
    setSoldMessage('An error occurred while marking the product as sold. Please try again.');
```

`finally {`

```
    setIsMarkingSold(false);
  }
};
```

Replace `window.location.reload()` (which reloads the entire SPA). This will put bad experience for users.

```

window.location.reload();
```

In the designing part, you can convert all of the design into sub components that would make it easier to understand and modify. Like `userIcon.jsx` etc.

```

{/* User Avatar */}
<div className="relative">
  <img
    src={contact.avatar}
    alt={contact.name}
    className="w-10 h-10 rounded-full object-cover"
  />
</div>
```

4.7.Review - chat.controller.js

- **Code Reviewer:** Karan Patel
- **Code Owner:** Hamza Butt
- **Date:** 27.06.2025
- **File to be reviewed:** Backend/controllers/chat.controller.js

Overview

This module handles core chat functionalities:

- Fetching user-specific chats (`getUserChats`)
- Fetching messages in a specific chat (`getChatMessages`)
- Deleting a chat for a user (`deleteChat`)

Code Features

- **Query logic:** The conditional Op.or and Op.and blocks in `getUserChats` are
- **Soft delete strategy:** Dual-flag logic for `owner_deleted` and `other_deleted` is good — allows for per-user soft delete while enabling actual purge if both users delete.
- **Error handling:** Consistent try/catch blocks and `res.success/res.error` pattern — keeps responses uniform.

Suggestion / Improvement

Missing Null Checks:

`json.product` could be null if somehow the relation is broken. You use:

```
// attach a single image_url and strip out mediafiles array
const result = chats.map(c => {
  const json = c.toJSON();
  const media = json.product.mediafiles || [];
  json.product.image_url = media.length ? media[0].file_path : null;
  delete json.product.mediafiles;
  return json;
});
```

Muhammad Hamza Butt, last week • update chat implementataion logic ...

Suggestion:

```
const media = json.product?.mediafiles || [];
```

Consistent Logging:

Can add some logs for `getUserChats` as well in case of debugging some issue related to fetching chats.

Hard-coded status code:

Instead of hardcoded status you can create a constant for more readability:

```
    return res.success('User chats fetched successfully', result, 200);
  } catch (err) {
    return res.error(err.message || 'Failed to fetch user chats', 500);
  }
};

exports.getChatMessages = async (req, res) => {
  try {
    const { chat_id } = req.query;

    const chatExists = await chat.findByPk(chat_id);
    if (!chatExists) {
      return res.success('No messages yet. Chat can be started.', [], 200);
    }
  }
}
```

Suggestion:

```
const HTTP_STATUS = {

  OK: 200,

  BAD_REQUEST: 400,

  FORBIDDEN: 403,

  NOT_FOUND: 404,

  INTERNAL_SERVER_ERROR: 500

};
```

Optimize Database Query:

Consider adding indexes and limiting data fetched, it will improve performance when there are a lot of messages to fetch.

```
const chats = await chat.findAll({
  attributes: ['chat_id', 'created_at', 'product_owner_id', 'other_person_id'],
  where: deletionFilter,
  include: [
    {
      model: productlisting,
      as: 'product',
      include: [{
        model: mediafile,
        as: 'mediafiles',
        attributes: ['file_path'],
        where: { is_approved: true },
        required: false
      }]
    },
    {
      model: message,
      as: 'lastMessage',
      attributes: ['message_id', 'content', 'sender_id', 'receiver_id', 'created_at']
    },
    { model: user, as: 'owner', attributes: ['id', 'username', 'email', 'createdAt'] },
    { model: user, as: 'otherPerson', attributes: ['id', 'username', 'email', 'createdAt'] }
  ],
  order: [['chat_id', 'DESC']]
});
```

Muhammad Hamza Butt, 4 weeks ago • work on structuring product listing response an...

5. Self-check on best practices for security

Major assets we are protecting:

- User accounts & credentials (email, password)
- Product listings and user-generated content
- Internal messaging data between buyers and sellers
- Admin dashboard & moderation tools
- Database and API endpoints

Major threats & our protections:

Asset	Threat	Protection
User accounts & passwords	Credential theft, brute force attacks	Passwords are hashed using bcrypt before storage in the database (confirm: we never store plain-text passwords).
Product listings & content	Injection attacks, spam, or malicious data	All form inputs (title, description, tags, etc.) are validated on both frontend and backend. Special characters are escaped in database queries to prevent SQL/NoSQL injection.
Internal messaging data	Data leakage, XSS	Messages are sanitized before storage and rendering. HTML/JS injection is prevented using output encoding.
Admin tools	Unauthorized access	Role-based access control checks (eg, admin@hs-fulda.de is configured in code and ideally moved to env config). Admin-only API routes are secured by middleware checking user roles.

Database/API	DDoS, unauthorized queries	Backend API uses JWT auth to validate logged-in users. Query rate limiting could be applied to endpoints. The DB is hosted in a private subnet (AWS RDS) and cannot be reached publicly.
--------------	----------------------------	--

Confirm encryption of passwords:

All user passwords are securely hashed with bcrypt before saving in the database. No plain-text passwords are stored.

Confirm input data validation:

- On *Create Listing* page: validate title, description, price, tags, images, category.
- *Search bar* input is validated to allow up to **40 alphanumeric characters only** to prevent injection and maintain performance.
- Numeric fields (eg, price) are validated to ensure they contain valid numbers within a reasonable range.

These validations are implemented in the React frontend and double-checked in the Node.js/Express backend before database writes.

6. Self-check: Adherence to original Non-functional specs

1. Application shall be developed, tested and deployed using tools and servers approved by Class CTO and as agreed in Milestone 0. Application delivery shall be from chosen cloud server (**Achieved**)
2. Application shall be optimized for standard desktop/laptop browsers e.g. must render correctly on the two latest versions of two major browsers. (**Achieved**)
3. All or selected application functions must render well on mobile devices. (**Achieved**)
4. Data shall be stored in the database on the team's deployment cloud server. (**Achieved**)
5. No more than 50 concurrent users shall be accessing the application at any time. (**Achieved**)
6. Privacy of users shall be protected and all privacy policies will be appropriately communicated to the users. (**Not Achieved**)
7. The language used shall be English (no localization needed). (**Achieved**)
8. Application shall be very easy to use and intuitive. (**Achieved**)
9. Application should follow established architecture patterns. (**Achieved**)
10. Application code and its repository shall be easy to inspect and maintain. (**Achieved**)
11. Google analytics shall be used (optional for Fulda teams) (**Not Achieved**)
12. No email clients shall be allowed. (**Achieved**)
13. Pay functionality, if any (e.g. paying for goods and services) shall not be implemented nor simulated in UI. (**Achieved**)
14. Site security: basic best practices shall be applied (as covered in the class) for main data items. (**Achieved**)
15. Application shall be media rich (images, video etc.). Media formats shall be standard as used in the market today. (**Achieved**)
16. Modern SE processes and practices shall be used as specified in the class, including collaborative and continuous SW development. (**Achieved**)

17. For code development and management, as well as documentation like formal milestones required in the class, each team shall use their own GitHub to be set-up by class instructors and started by each team during Milestone 0. (**Achieved**)
18. The application UI (WWW and mobile) shall prominently display the following exact text on all pages "Fulda University of Applied Sciences Software Engineering Project, Summer 2025 For Demonstration Only" at the top of the WWW page. (Important so as to not confuse this with a real application). (**Achieved**)