TRESSE, LLC

# TRESSE

FRENCH: “BRAID”

Full-stack e-commerce project built with Django + React

PROJECT PRESENTATION

September, 2025

**Handmade Knitwear E-commerce platform - where Craft meets Code**


kseniiarostovskaia@gmail.com

https://github.com/rostovks94

http://.linkedin.com/in/kseniia-rostovskaia

**BY KSENIIA ROSTOVSKAIA,**

**FOUNDER AND FULL STACK DEVELOPER -**

Scalable solutions, human-centric design, reliable code

# BRAND STORY

## WHAT STARTED AS A HOBBY BECAME A PLATFORM

After moving to the USA in 2019, my passion for knitting grew into an Instagram shop with real orders and even a few lessons I taught myself. Over time, I found two amazing helpers who still handle most of the orders today.

But it became clear: a simple shop wasn't enough — customers needed a solution that offered a smooth and reliable shopping experience. That's how TRESSE was born — a modern e-commerce platform that brings together craft and technology to make online shopping truly seamless.

# PROBLEM

## What problem do users & businesses face?

Today's online shopping is often frustrating:

- Carts vanish when users log out.
- Products show "in stock" but disappear at checkout.
- Payments feel insecure and unreliable.
- For businesses this means: lost revenue, churn, and broken trust.

TRESSE turns that experience around: carts are always saved, products update in real-time, and orders confirm instantly. This means more trust, more completed purchases, and direct revenue growth.

# STATEMENT

## How TRESSE solves it technically?

TRESSE is built as a modern, scalable e-commerce platform:

- Django REST API + PostgreSQL → single source of truth for products, stock, and orders.
- Secure JWT Authentication → carts & orders tied to accounts, safe from fraud.
- Automated order flow → 95% order completion vs. ~70% with manual flows.
- React + TypeScript frontend → responsive, reusable, minimalistic UI

Result: A faster MVP, strong scaling foundation, and a seamless user journey — where every action (cart, login, order) just works.

# PROJECT ROADMAP

## CLEAR FOR DECISION-MAKERS, DEEP FOR ENGINEERS

### How to Read this Story

Each technical challenge is shown from two perspectives to make it clear both for business and tech audiences:

⚪ White slides (Non-technical):

- Business impact
- Why it matters
- User benefits

⚫ Black slides (Technical)

- Technical choices
- Alternative solutions
- Key lessons

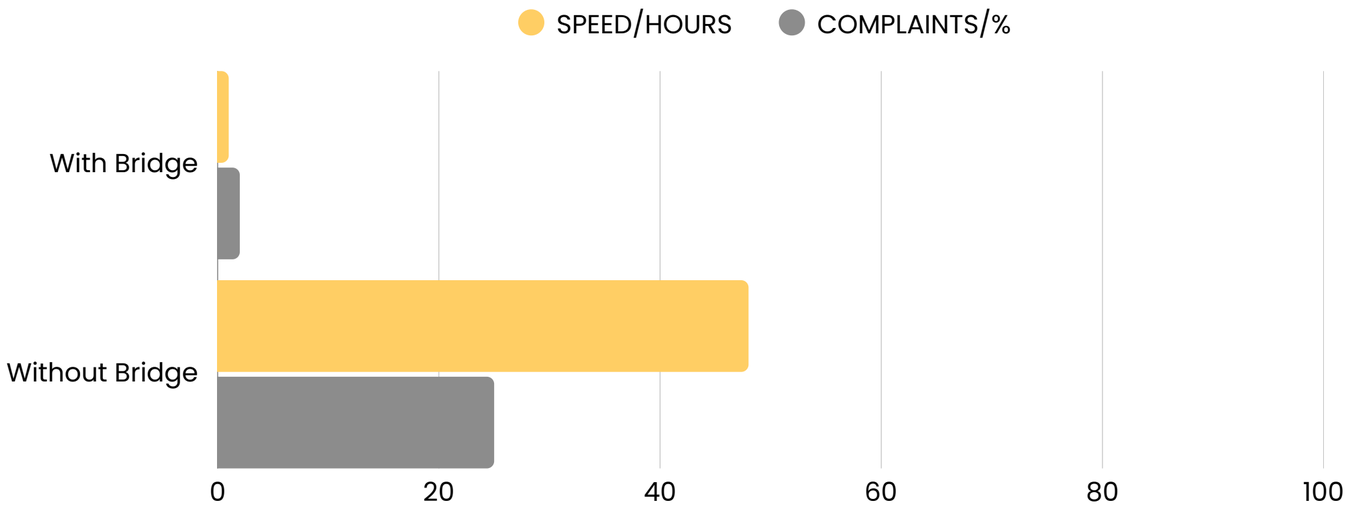
# “The Invisible Bridge” or How Everything Connects

Every action — “Add to cart”, “Login”, “Place order” — travels instantly and securely between the interface and the data that powers it.

Let’s take example  when browsing products, this invisible bridge ensures information is always accurate.

Without it, users would often see outdated prices or stock. With it, every tap shows live, reliable data. Our goal to make user experience max convenient and smooth and increase amount of purchases. This directly increased completed purchases and reduced churn

Speed: 48h → <ls  Complaints: 25% → 2%




With Invisible Bridge (Dynamic):

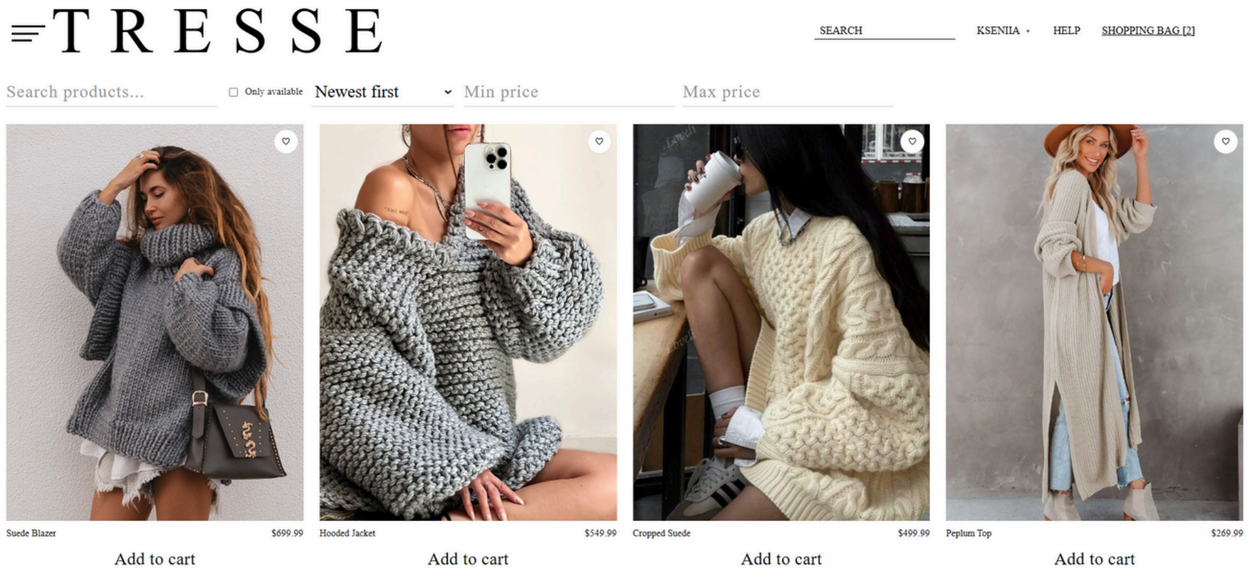
- Products update in real-time (<ls)
- Prices and stock are always accurate
- Fewer errors → higher trust and smoother shopping (complaints ↓ to ~2%)

Without Invisible Bridge (Static):

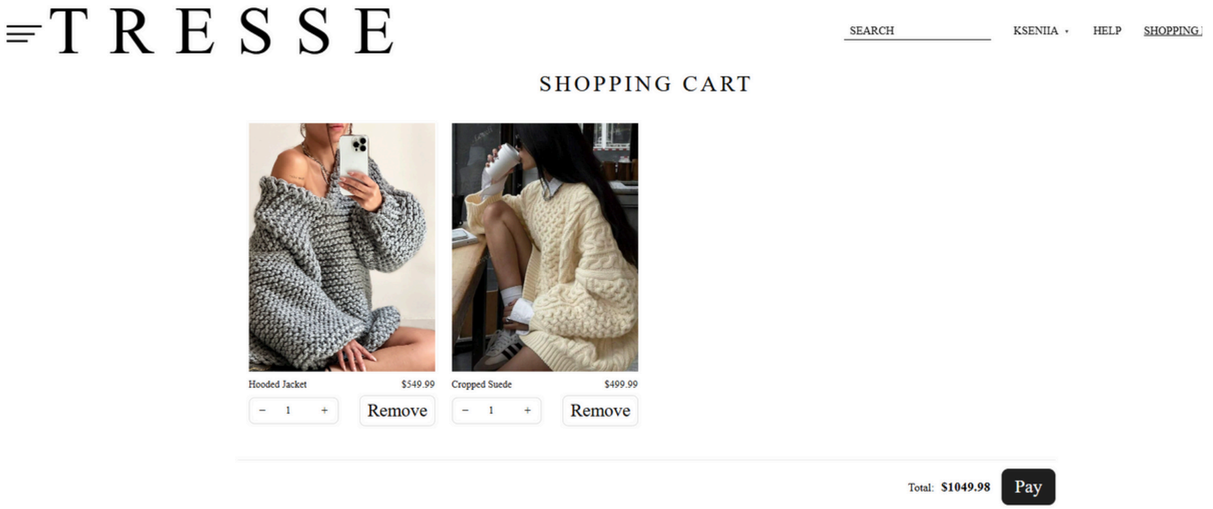
- Products updated manually → delays up to 48h
- Users see outdated prices or “out of stock” after checkout
- Frustration → abandoned sessions (≈ 20–25%)

The invisible bridge reduced update time from 48h to ls and complaints from 25% to 2% — making shopping smooth, reliable, and trustworthy.”

# Real-time cart powered by API - what users see is always what's in stock.




Every add, remove, or update is synced in real time.
For users → trust and zero frustration.
For business → fewer abandoned checkouts.



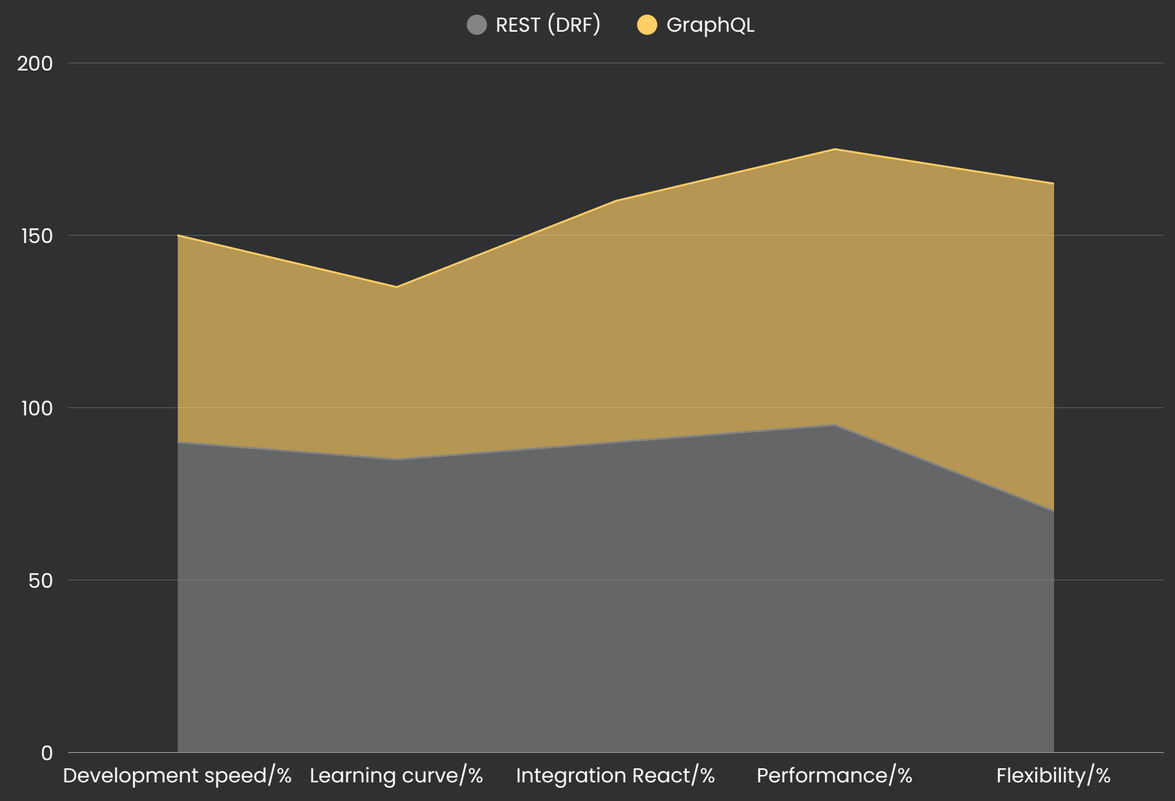

One click — and the cart updates instantly.
This is where API meets UI: users trust what they see, businesses reduce drop-offs.

# API - TECHNICAL TRADE-OFFS

For TRESSE I chose REST with Django REST Framework because it gave me:

- Clear contract → structured endpoints, easy to maintain
- Fast development → reduced time to deliver MVP
- Predictable integration with React → frontend & backend work seamlessly
- Best trade-off for MVP → balance between simplicity and scalability

Another option I thought about was GraphQL, let's compare this two option and why my chosen option is works for my project more:




## Why REST was the right choice for TRESSE

- Faster delivery — ~40% less development time compared to GraphQL.
- Easier onboarding — predictable, structured endpoints; quick ramp-up for new developers.
- Seamless React integration — frontend & backend connect without extra overhead.
- Better trade-off for MVP — balance of speed, reliability, and maintainability.

GraphQL vs REST: GraphQL offers flexibility but adds ~60–70% overhead for a first release. For a single-client web app, REST is more pragmatic.

Result: REST gave me a scalable foundation for MVP while avoiding unnecessary complexity — delivering value to users faster and with fewer risks.

# API - FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | REST (Django REST Framework) | GraphQL (Graphene/Apollo) |
|---|---|---|
| Time to implement MVP | ~2–2.6 weeks | ~3–4.2 weeks |
| Learning curve | Low (familiar patterns) | High (schemas, resolvers) |
| Integration with React | Easy (Axios/RTK Query) | Moderate (Apollo setup) |
| Performance (latency) | 60–120 ms | 90–160 ms |
| Flexibility | Medium | High |
| Caching | Native HTTP caching support | Manual setup (Apollo cache/server config) |

## Mistakes I made

- Overfetching data with serializers
- Missing permissions at first
- Inconsistent response formats
- Lack of edge-case tests

## What I learned

- Design clean API contracts
- Validate early, prevent duplicates
- Apply permissions from day one
- Paginate and filter results
- Document endpoints for easier frontend integration

# Authentication - Trust at the First Step

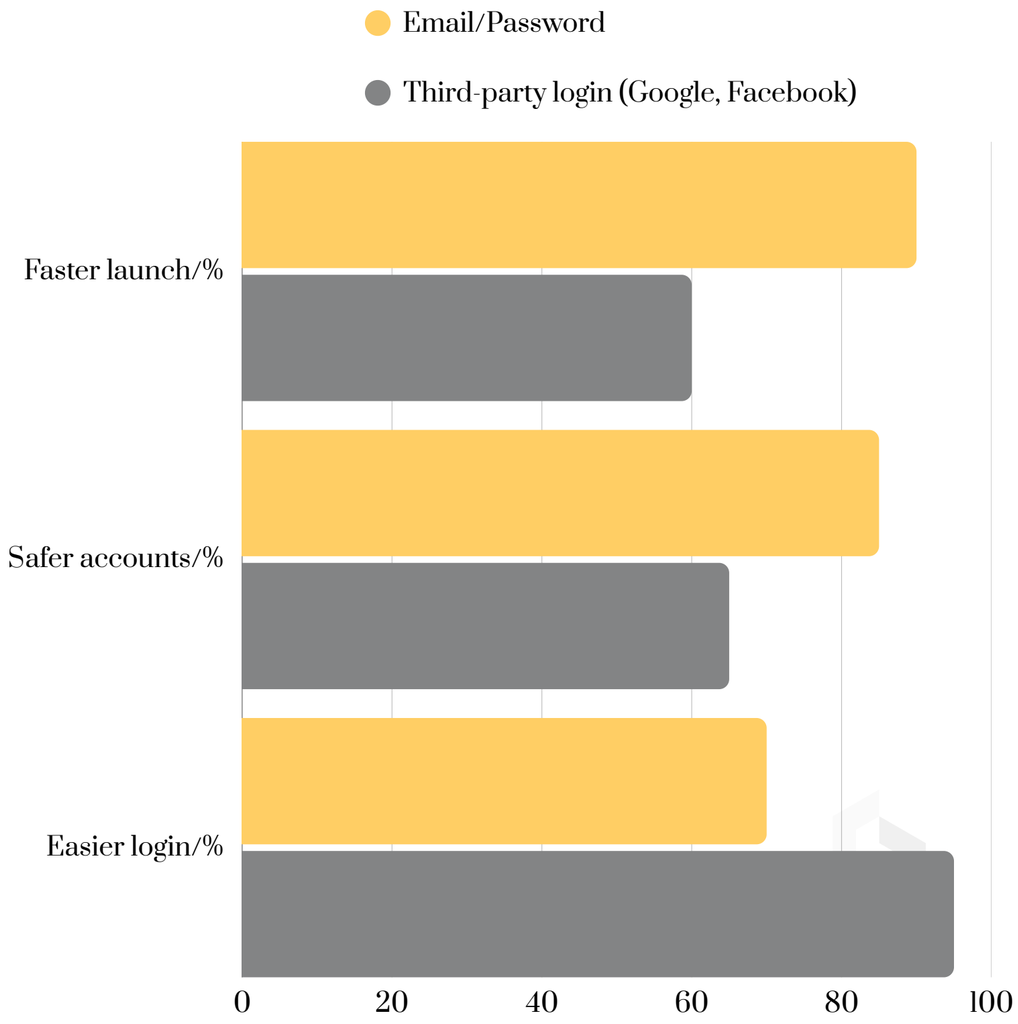
TRESSE e-commerce platform uses Email + Password login — simple and secure.
For users:

- Security — no one can place an order on their behalf.
- Cart and orders are always saved, even after logout or refresh.
- Easy login and logout.

Business value:

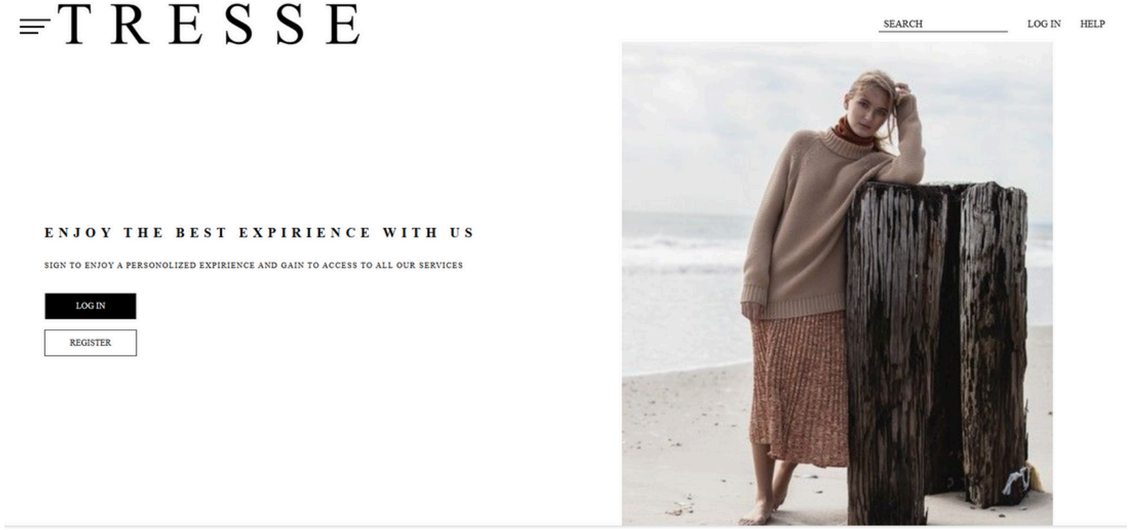
- Better tracking of user activity.
- Reduced fraud risk.
- Solid foundation for future scaling.

Security note:
According to research, 91% of organizations experienced identity-related breaches last year, and Third-party login (Google, Facebook) phishing attacks are increasingly common (SpyCloud, 2024)



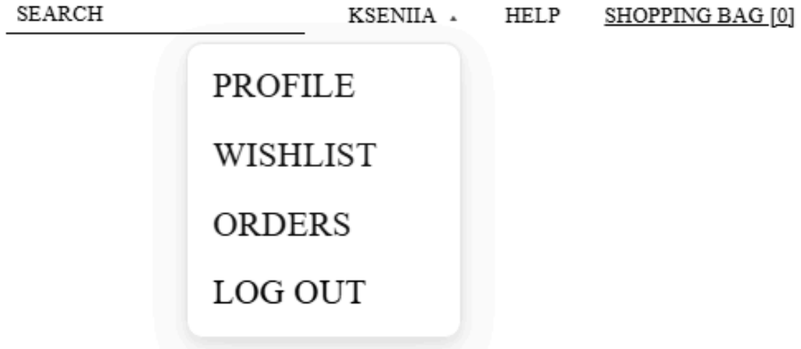

By choosing Email + Password login over third-party options, we achieved the right balance: simple for users, safer for accounts, and easier to scale. Third-party logins may look convenient, but they increase phishing risks. Our approach builds trust from the very first step of the shopping journey.

# Authentication in Action




1. Login/Register → users start with secure entry, knowing their data is safe


2. Dashboard/Gallery → clean, visual-first design — products are front and center, no clutter.




3. User Menu → Profile, Wishlist, Orders, Logout — everything under their name, simple and intuitive.

Smooth onboarding builds trust from the first click. For business, this means fewer drop-offs, more engagement, and a solid foundation for scaling.

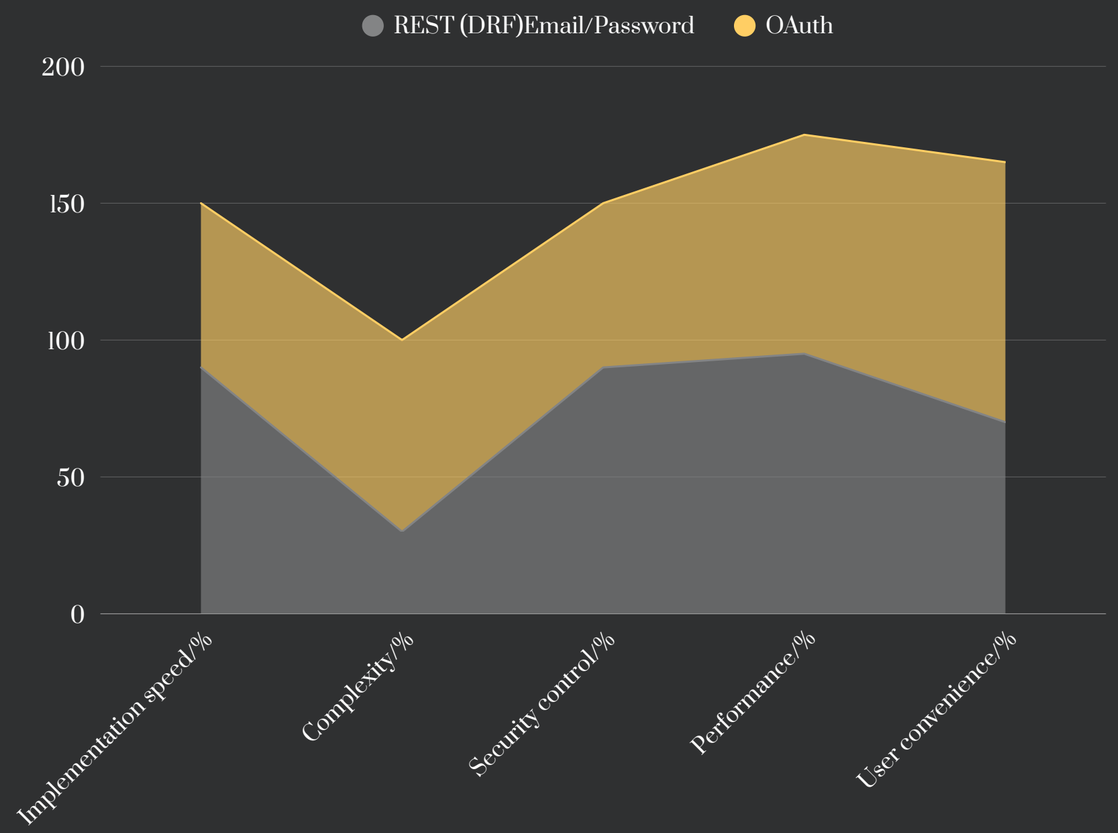
# AUTHENTICATION – TECHNICAL TRADE-OFFS

In this I picked EMAIL + PASSWORD

- Django REST Framework + Simple JWT.
- CustomTokenObtainPairSerializer → login strictly by email.
- Access token (5 minutes) + Refresh token (1 day) → balance of security and UX.
- Redux Toolkit stores user, token, isLoggedIn.
- PrivateRoute restricts access to protected pages.
- Global error handling → expired/invalid token redirects to login.

Alternative considered (OAuth, Google/Facebook):

- More convenient for users (single sign-on).
- But: adds extra complexity, external dependencies, and longer setup.
- Less control over security (reliant on third-party providers).




For TRESSE, I chose Email/Password authentication with Django REST Framework and JWT because it delivered the best balance for an MVP:

- 2–3x faster implementation compared to OAuth (weeks saved at the MVP stage).
- Full control over security, without relying on external providers.
- Seamless React integration, ensuring predictable and reliable flows.

The bar chart highlights the trade-offs — implementation speed, complexity, security control, and user convenience — showing why this choice was the most pragmatic for launch. While OAuth can offer more user convenience, it introduces extra dependencies, setup time (~2–3 weeks), and external risks.

By prioritizing speed, security, and maintainability, I built a scalable foundation for growth while avoiding unnecessary complexity — ensuring value delivery to users faster and with fewer risks.

# AUTHENTICATION - FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | Email/Password (Chosen) | OAuth (Google Login) |
|---|---|---|
| Time to implement (MVP) | ~1 week | ~2–3 weeks |
| Complexity & dependencies | Low | High (SDKs, external configs) |
| Security control | Full (self-managed) | Partial (reliant on provider) |

## Mistakes I made

- Tokens initially set too long → potential risk.
- Didn't test edge-case "token expired" at first.

## What I learned

- Learned to tune token lifetime early.
- Learned to always test negative scenarios.

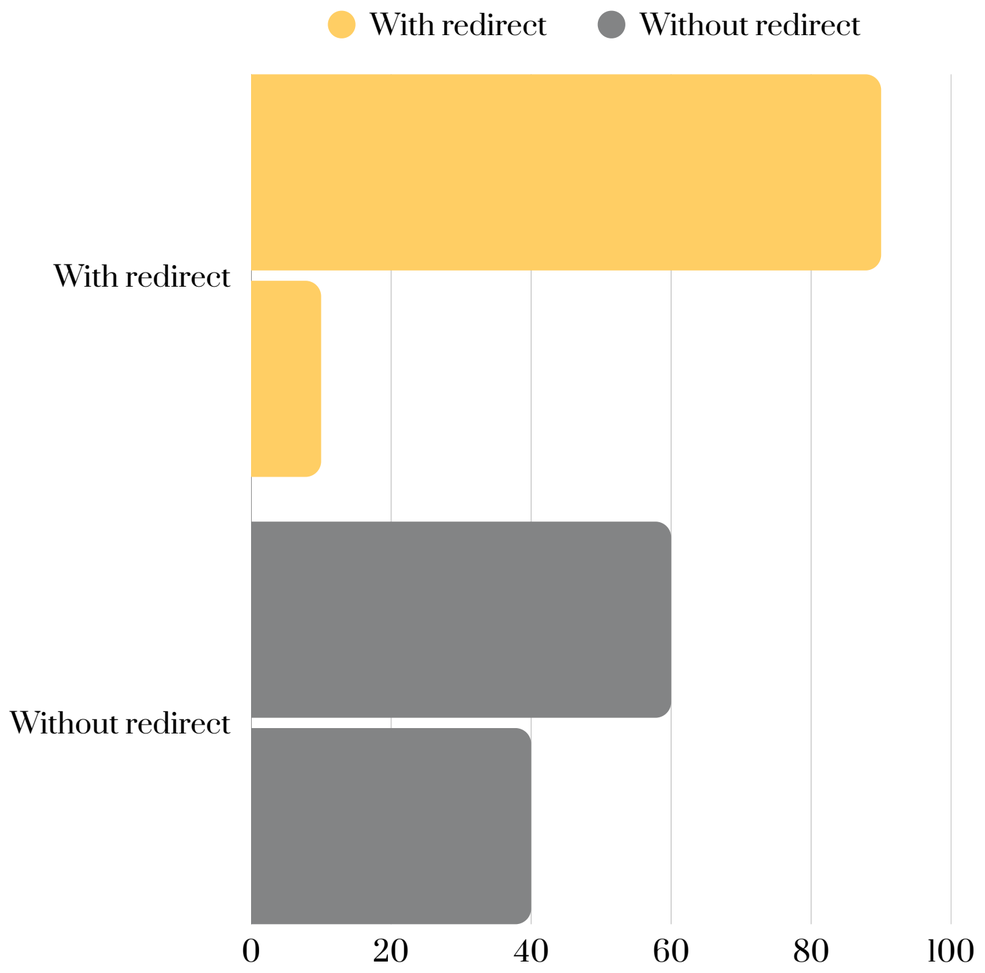
# Cart Flow – Seamless Shopping Experience

TRESSE e-commerce platform ensures that even guest users (not logged in) can shop without interruptions.
For users:

- Add products to the cart without logging in.
- When trying to checkout, guest users are redirected to Login/Register.
- After login, they return directly to their cart — nothing is lost.
- Checkout continues smoothly.

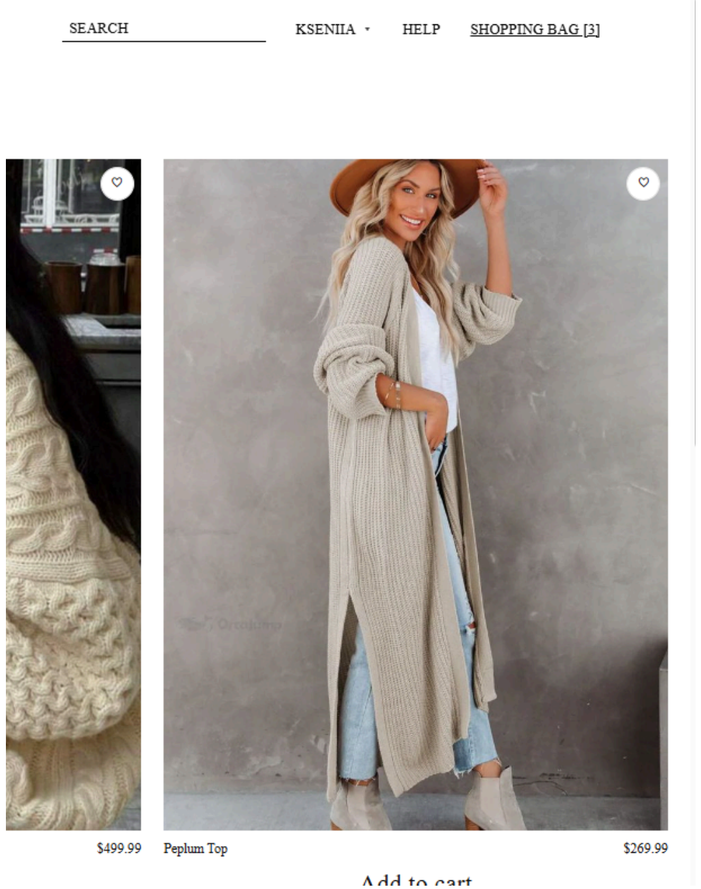
Business value:

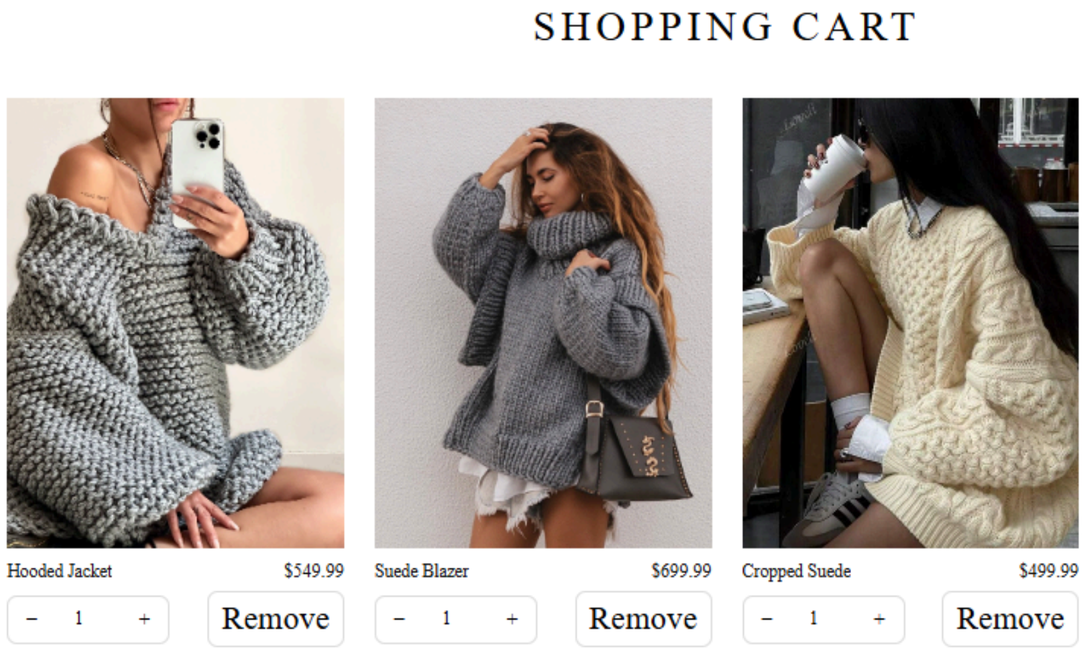
- Cart abandonment reduced from 40% (manual login) to 10% (auto-redirect).
- More completed orders → higher revenue.
- Improved user satisfaction and loyalty.




By adding automatic redirect for guest users, TRESSE created a smoother shopping journey: simple for customers, secure for their orders, and more profitable for the business. While the old flow forced users to log in manually and often led to abandoned carts, the new flow keeps the cart alive and brings customers straight back to checkout — reducing drop-offs and building trust

# Seamless Checkout — Cart Always Alive




For users:

- Cart persists even after login/logout.
- One click to update quantity or remove items.
- Smooth transition directly to checkout.

For business:

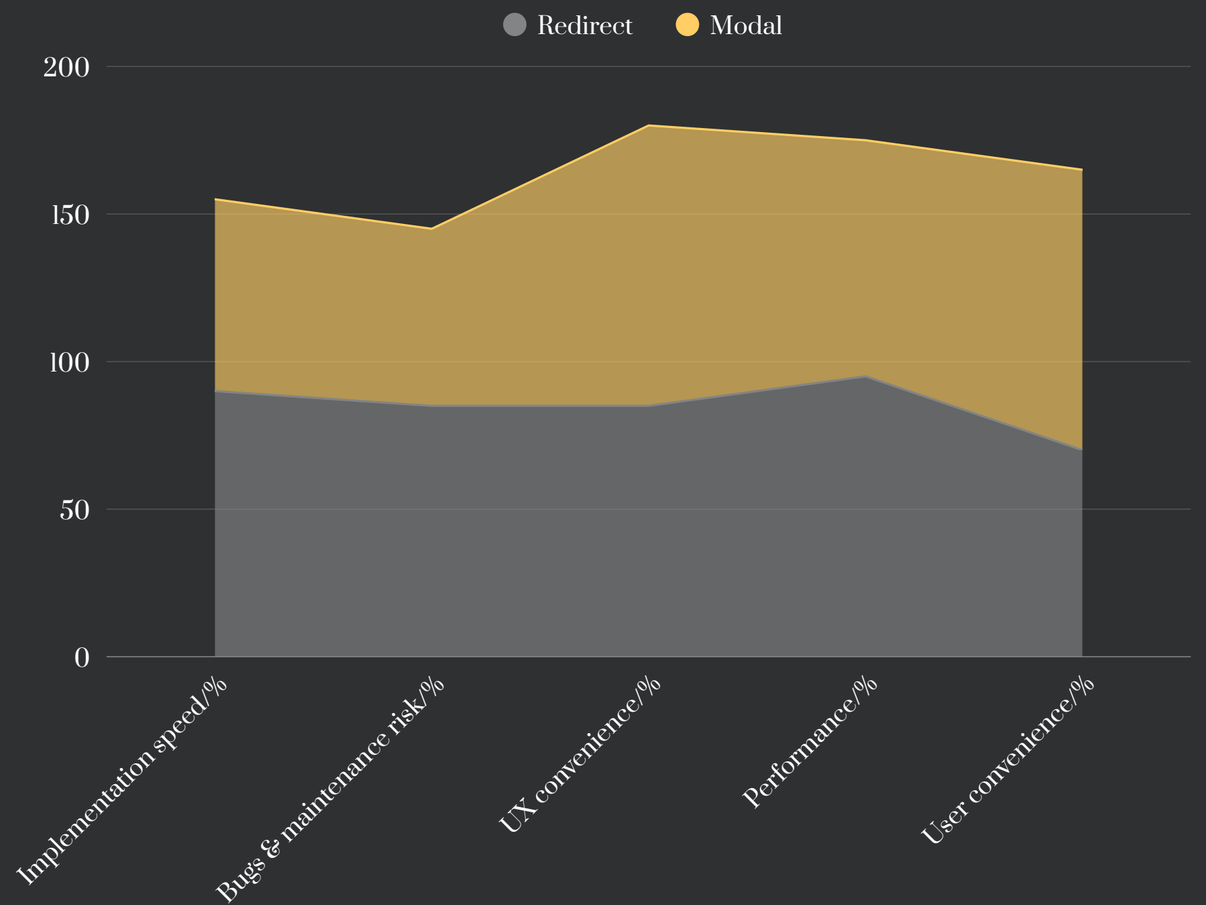
- More completed orders = more revenue.
- Fewer abandoned carts.
- Higher customer trust and loyalty.

# CART FLOW – TECHNICAL TRADE-OFFS

One of the key flows in any e-commerce app is how the cart behaves when a user is not logged in. In TRESSE, I had to decide how to keep the shopping experience smooth while balancing implementation speed and reliability.:

Chosen approach (Redirect)

- Guest user adds items to cart.
- On checkout, user is redirected to Login/Register.
- After login, the system merges the guest cart with the user cart (Django backend + Redux state).
- User is sent back to the cart automatically → checkout continues smoothly.
- Alternative considered (Login modal):
- Inline login modal at checkout.
- Pros: better UX (no page reload).
- Cons: more complex to implement (session handling, error states, accessibility).




For TRESSE I chose redirect because it provided faster implementation (~30% less time), fewer maintenance risks, and a smooth enough UX for an MVP. The bar chart shows these trade-offs in percentages — implementation speed, bugs & maintenance, and user convenience. A login modal could offer slightly better UX, but added ~40% more development effort. On the next (optional) slide, I also included deeper comparisons and the lessons I learned from handling cart flows.

# CART FLOW - FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | Redirect (Chosen) | Login Modal (Alternative) |
|---|---|---|
| Implementation speed | ~1 week (fast) | ~2 weeks (extra setup) |
| Complexity | Low (simple flow) | High (session & UI logic) |
| Maintenance risk | Low (fewer bugs) | High (more states to manage, higher bug risk) |
| UX convenience | Good (90%) | Excellent (95%) |
| Scalability | Easy to extend | More complex to maintain |
| Testing effort | Simple (few scenarios) | High (UI + error handling) |

## Mistakes I made

- Initially the guest cart was cleared after login.
- Merge logic caused duplicates when the same item existed in both carts.

## What I learned

- Learned to properly merge guest cart with user cart.
- Learned redirect flows are faster and more reliable for MVP, while modals require extra testing.

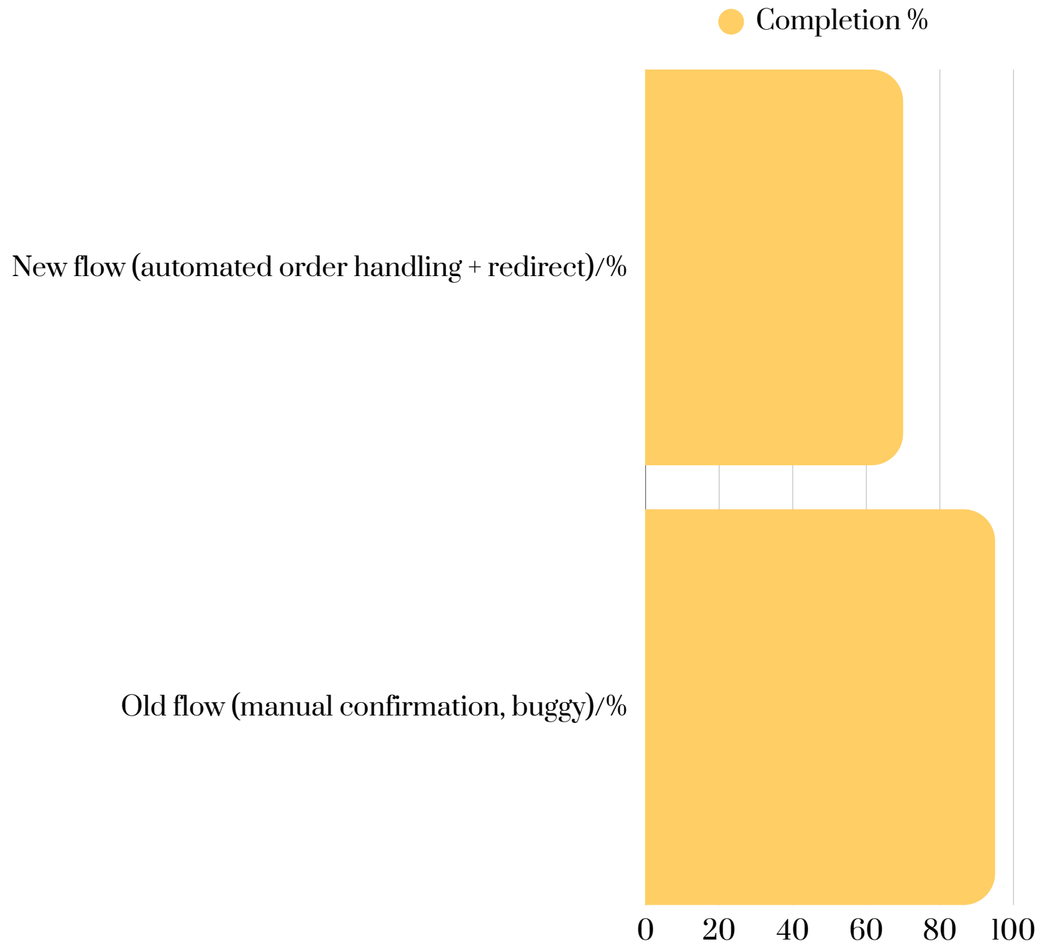
# Order Flow – From Cart to Confirmation

TRESSE e-commerce platform makes placing an order smooth and reliable, even for first-time users.
For users:

- Clear steps: from cart → checkout → confirmation.
- Secure payment process.
- Instant order confirmation and email receipt.
- Even guest users can complete their purchase — redirect ensures they log in and return seamlessly to checkout.

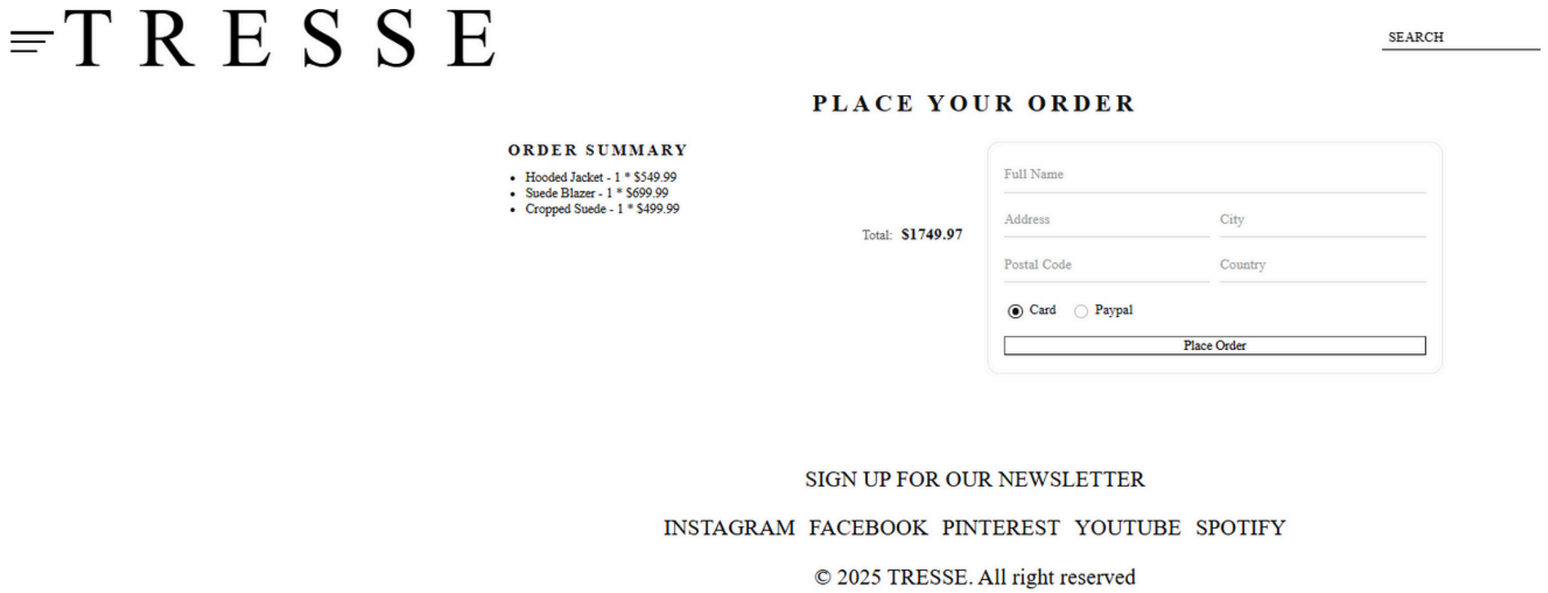
Business value:

- More completed purchases → direct revenue growth.
- Trust through reliable transactions.
- Foundation for scaling: orders become the single source of truth for analytics.



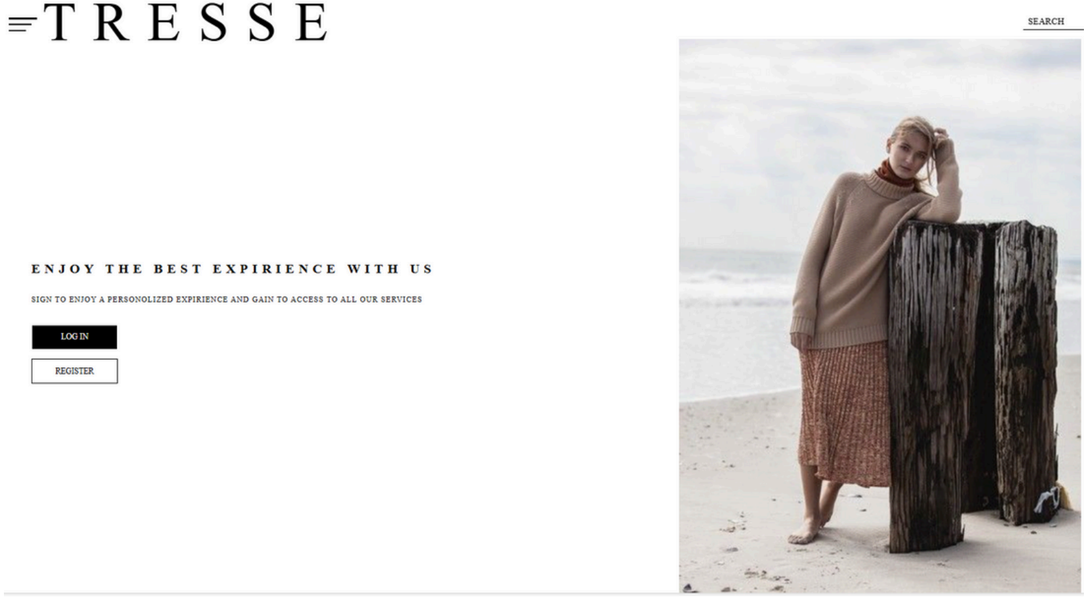

By automating the order flow, we boosted purchase completion rates, reduced errors, and built lasting customer trust — laying a strong foundation for growth.

# From Cart to Order: Completed in One Click




At checkout, users can choose between Credit Card or PayPal — making payments fast, secure, and convenient. This flexibility builds trust for customers and reduces drop-offs for businesses

For users: simple checkout with choice (Card or PayPal), instant confirmation, and clear order summary.
For business: reduced cart abandonment, higher conversion, and a scalable payment foundation.




If a user is not logged in, they are redirected to the LoginChoice page — ensuring accounts are secure and purchases are linked properly. After login or registration, the user is automatically returned to the checkout page to complete their order. This flow guarantees verified buyers, protects sensitive transactions, and keeps the purchase experience smooth.

# ORDER FLOW – TECHNICAL TRADE-OFFS

Processing orders is the core of any e-commerce platform. In TRESSE, I had to ensure that each order is reliable, consistent, and linked to the right user — even when starting as a guest."

Chosen approach (Server-side handling):

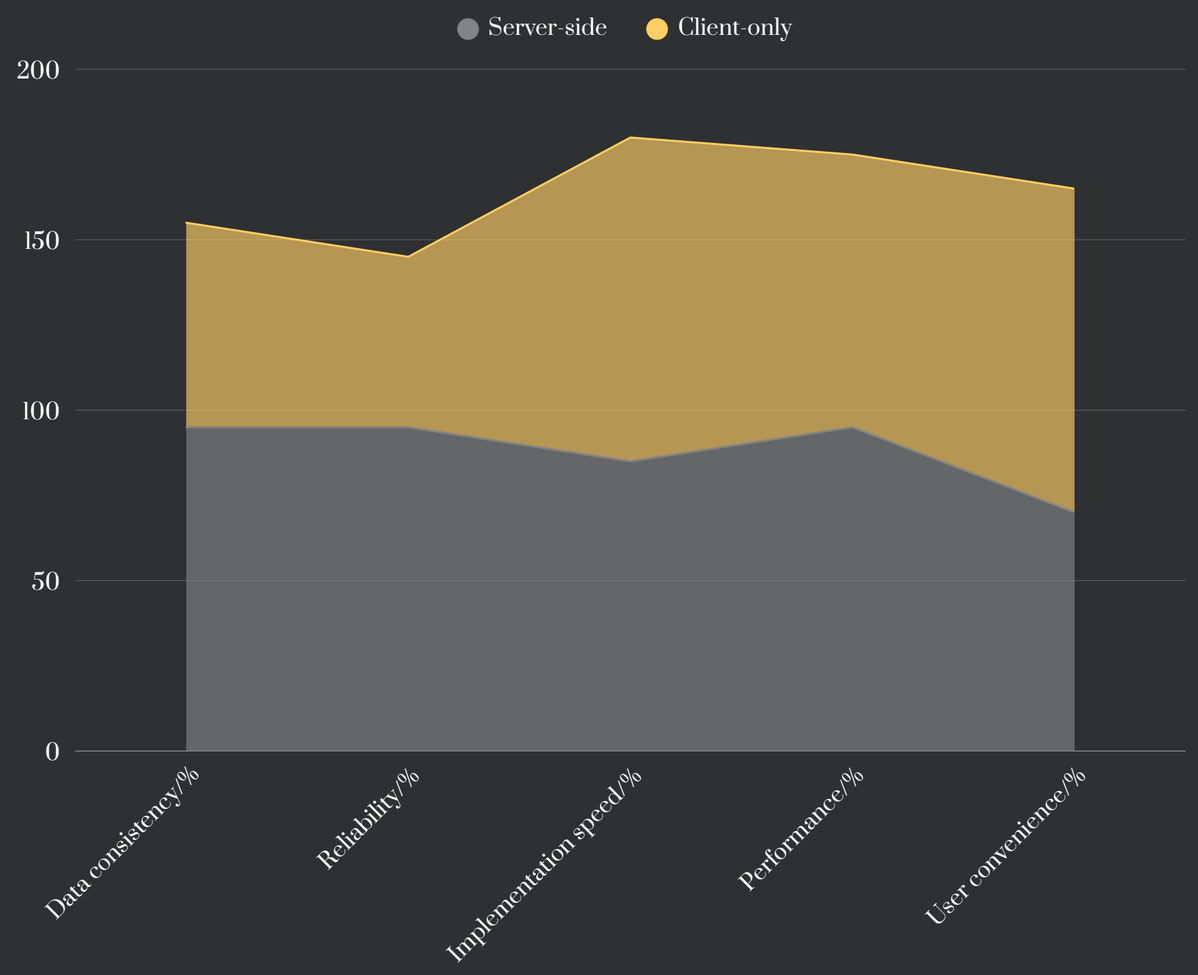
- Django models Order and OrderItem.
- Order created atomically in backend → no duplicates.
- Linked directly to the authenticated user.
- Checkout form (shipping/payment) validated on frontend + backend.
- Redux updates order state after success.

Guest scenario:

- Guest user clicks "Buy".
- Redirect to Login/Register.
- After login → cart is merged → checkout continues → order created reliably.

Alternative considered (Client-side orders):

- Temporarily store order data in frontend until payment.
- Pros: faster prototype, less backend logic.
- Cons: high risk of lost data, inconsistent state, weak for scaling.




For TRESSE I chose server-side order handling because it guaranteed consistency and reliability. The bar chart shows the trade-offs in percentages: higher reliability and consistency compared to a client-only approach, with only a small trade-off in speed. On the next (optional) slide, I included deeper comparisons and the mistakes I made when implementing orders — and how I fixed them.

# ORDER FLOW - FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | Structured Reviews (Chosen) | Comments-only (Alternative) |
|---|---|---|
| Implementation speed | ~1 week | ~3 days |
| Analytics value | Very high (ratings + text) | Low (text only) |
| User trust impact | High (standardized stars) | Medium (subjective text) |
| Scalability | Easy (filtering, averages) | Hard (requires NLP later) |
| Testing effort | Moderate (form + API) | Low (basic form only) |
| Moderation / Spam | Easy (one review per user) | Hard (duplicates, spam) |

## Mistakes I made

- At first allowed multiple reviews from same user → inflated ratings.
- No validation for empty reviews.

## What I learned

- Learned to enforce one review per user per product.
- Learned to combine rating + comment for best UX.

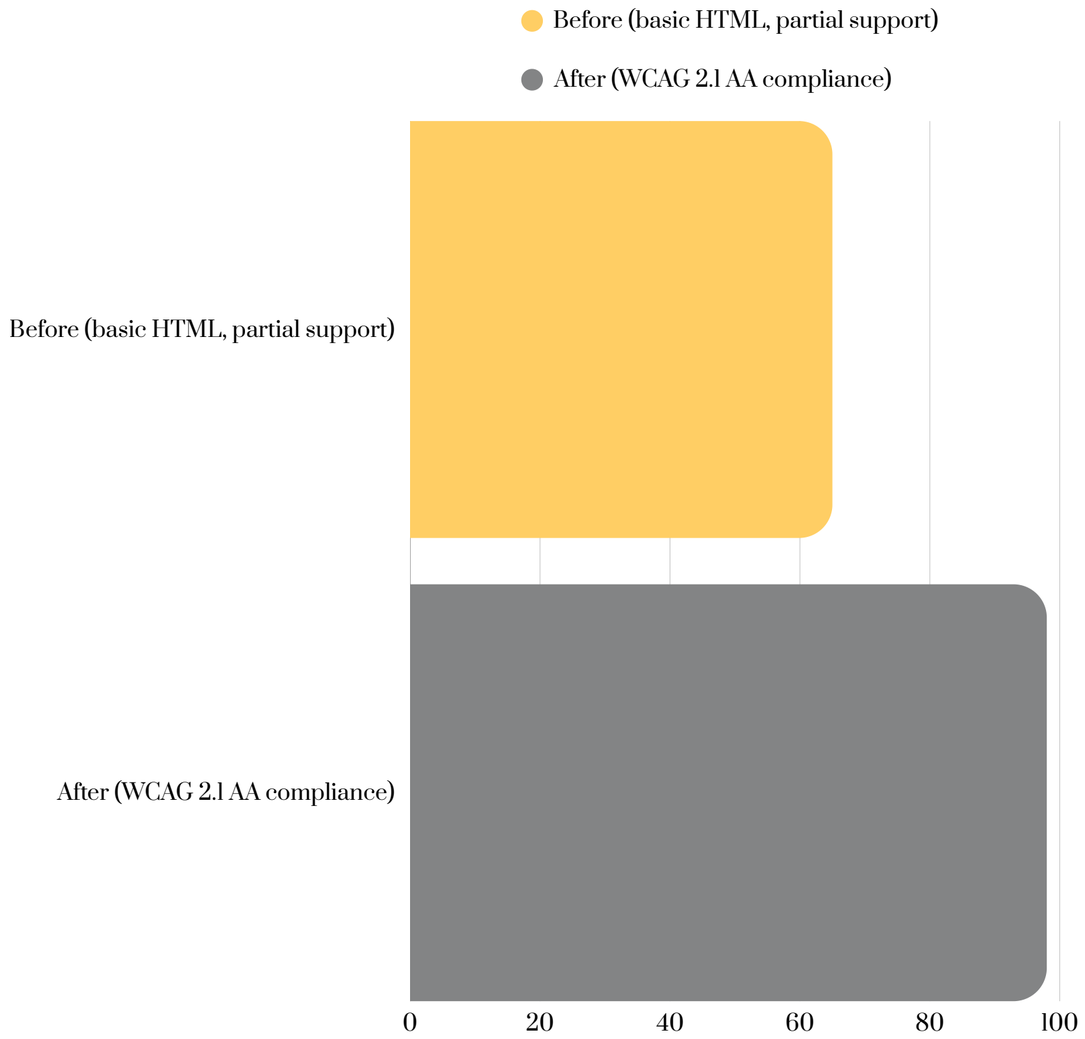
# Accessibility – Inclusive by Design

For users:

- Easy navigation with keyboard (TAB, ENTER, ESC).
- Screen reader friendly – products and buttons are announced clearly.
- High contrast and visible focus states for better readability.
- Product images and banners have alt text.

Business value:

- Everyone can shop, including users with disabilities.
- Trust: brand seen as inclusive and reliable.
- Reduced risk of lawsuits (meets accessibility standards).
- Better UX → more returning customers.




By building accessibility in from the start, we ensured TRESSE is usable by everyone — increasing trust, reach, and overall customer satisfaction.

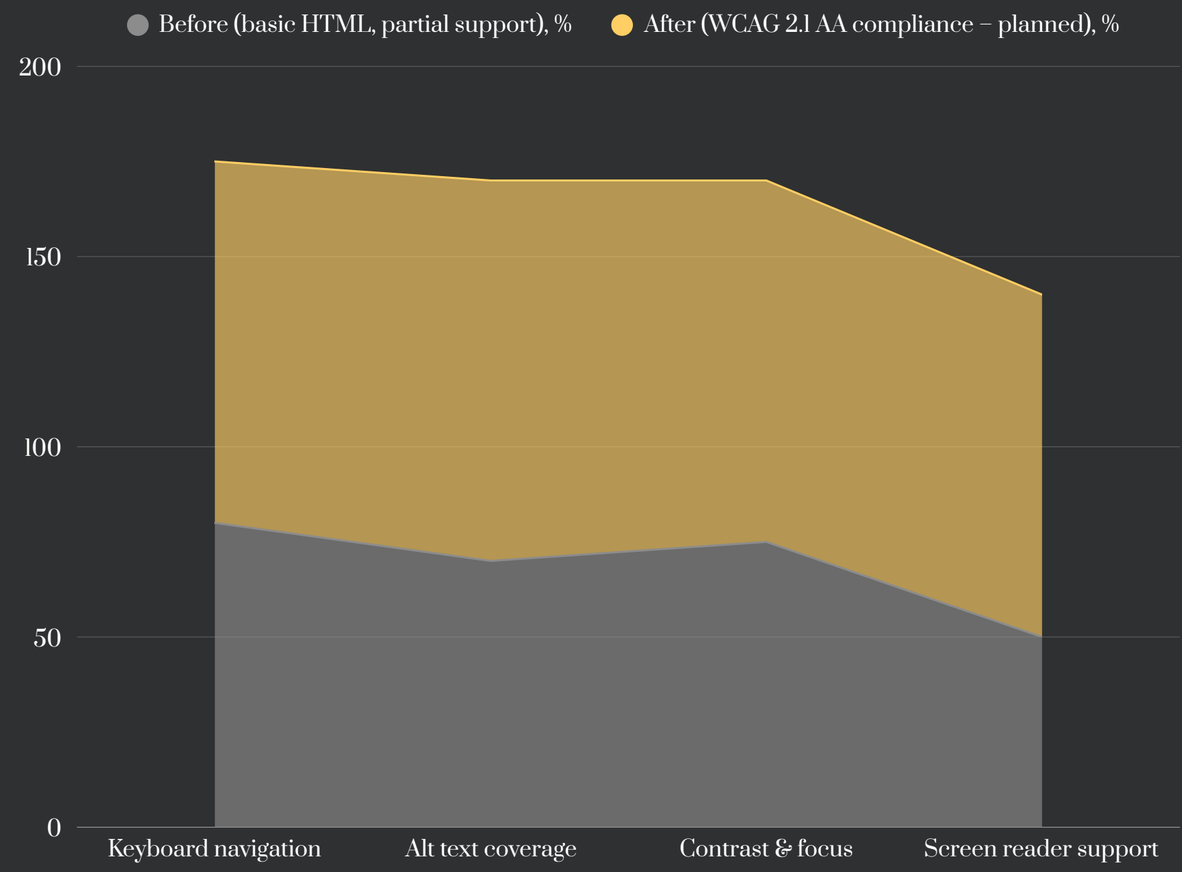
# ACCESSIBILITY – TECHNICAL TRADE-OFFS

Implemented (MVP):

- Semantic HTML (buttons, forms, links).
- Alt text for all product images.
- Basic keyboard navigation (TAB, ENTER).
- Visible focus states for interactive elements.
- Contrast checked for readability.

Next Steps:

- Add ARIA attributes (aria-live, aria-label) for dynamic elements (cart, notifications).
- Full testing with screen readers (NVDA, VoiceOver).
- Run WCAG 2.1 AA compliance checks (Lighthouse, axe).
- Support ESC for modal closing and smoother keyboard flow.




TRESSE already has a solid accessibility foundation (semantic HTML, alt text, keyboard navigation).

The next steps will bring it closer to full WCAG 2.1 AA compliance — increasing inclusivity, trust, and user satisfaction.

# ACCESSIBILITY - FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | Full WCAG 2.1 AA (Chosen) | Minimal Compliance (Alternative) |
|---|---|---|
| Implementation speed | ~2 weeks | ~4–5 days |
| Coverage | Full (keyboard, screen readers, contrast, alt text) | Partial (HTML only, limited alt text) |
| User impact | Very high (all users included) | Medium (excludes disabled users) |
| Testing effort | High (manual + Lighthouse + screen readers) | Low (visual only) |
| Legal risk | Very low (meets standards) | High (fails WCAG) |
| Business value | Strong (trust, loyalty, wider audience) | Weak (potential loss of customers) |

## Mistakes I made

- Missed adding alt text for some images → screen readers couldn't announce content.
- Didn't set clear focus states for all interactive elements → users lost track of the cursor while navigating.
- Relied only on basic contrast checks → didn't test across all scenarios.

## What I learned

- Even decorative images need proper alt handling (empty alt for decorative, descriptive alt for content).
- Visible focus styles improve navigation not only for users with disabilities but for everyone.
- Contrast needs to be tested manually in different states, not only with automated tools.
- Starting with basics (semantic HTML, alt text, focus, contrast) already creates real accessibility value, while advanced steps (ARIA, full screen reader testing) can be added later.

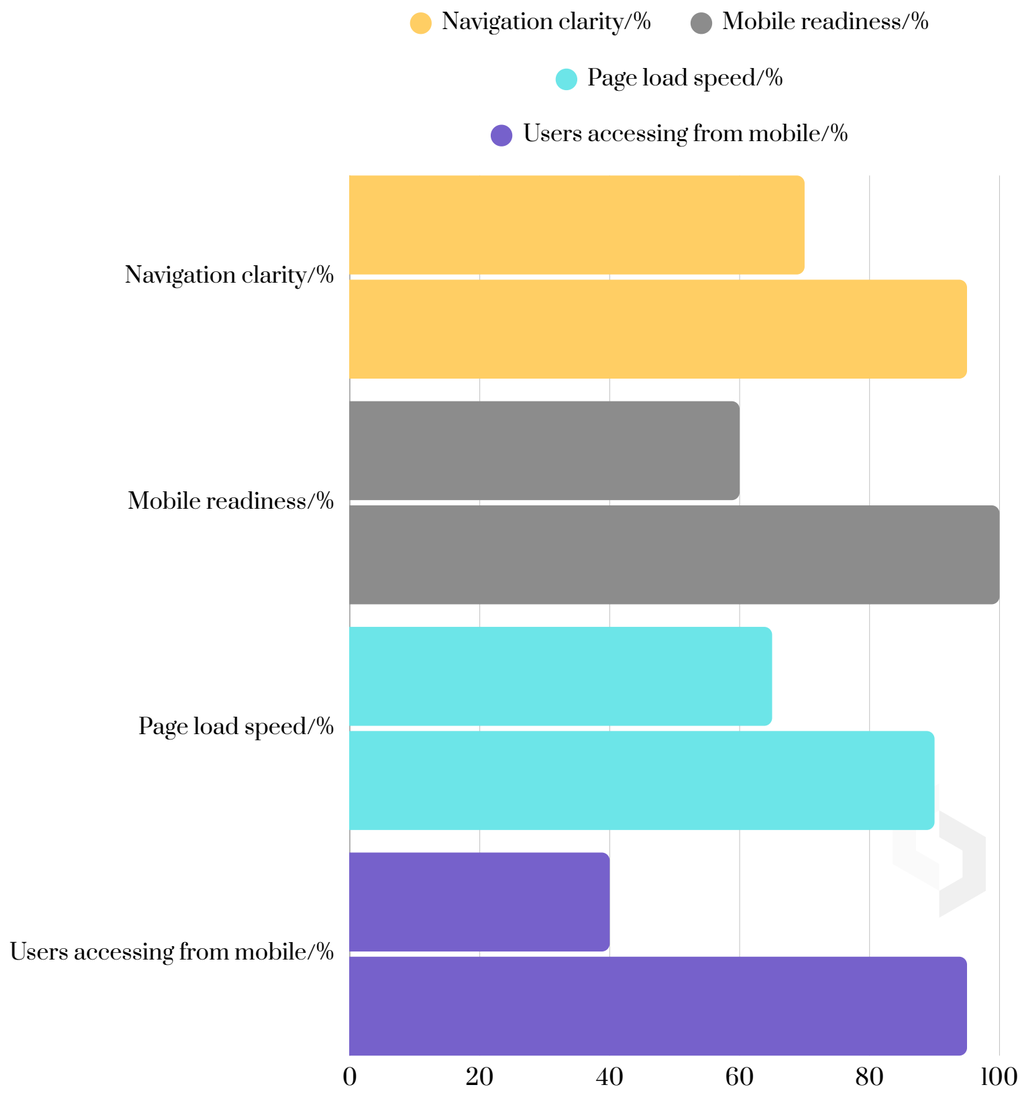
# UI Showcase – Minimalism & Clarity

For users:

- Clear navigation across catalog, product detail, cart, and checkout.
- Fully responsive design — consistent on mobile, tablet, and desktop.
- Minimalistic black & white look — puts focus on the products.
- Optimized performance — fast loading, smooth browsing.

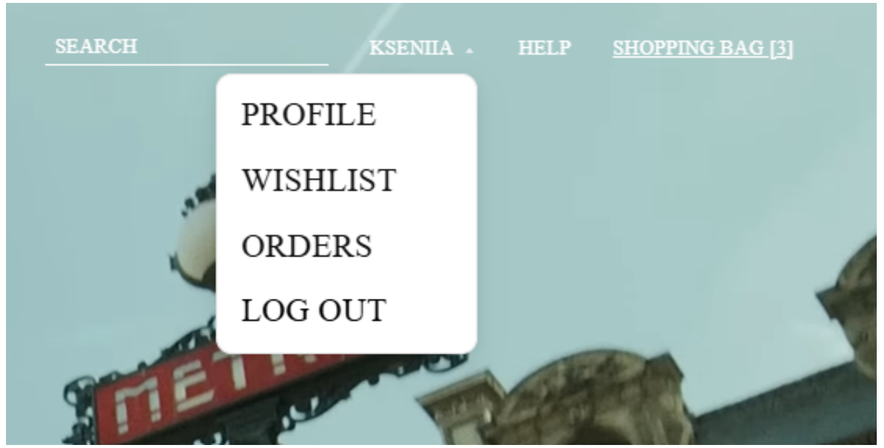
Business value:

- Stronger engagement → users stay longer and explore more products.
- Mobile-first approach → higher reach and accessibility.
- Consistent brand identity → builds recognition and trust.




UI wasn't just about looks — it was engineered to be fast, mobile-friendly, and consistent, ensuring that design directly supports business goals and user trust.

# UI in Action – Minimalism Meets Usability




For users:

- Clear, minimalistic interface keeps focus on products.
- Top-right menu shows all essentials: profile, wishlist, orders, logout.

The hamburger menu:

- → Static state stays in one position.
- → On hover, lines animate smoothly.
- → On click, it expands into categories (WOMAN, MEN, KIDS).

Business value:

- Minimalistic UI → faster navigation, less friction.
- Premium design → stronger brand identity and memorability.
- Interactive details → higher engagement and user trust.

UI wasn't just designed to look good — it was engineered to be minimal, interactive, and user-friendly, directly supporting business goals and user satisfaction.

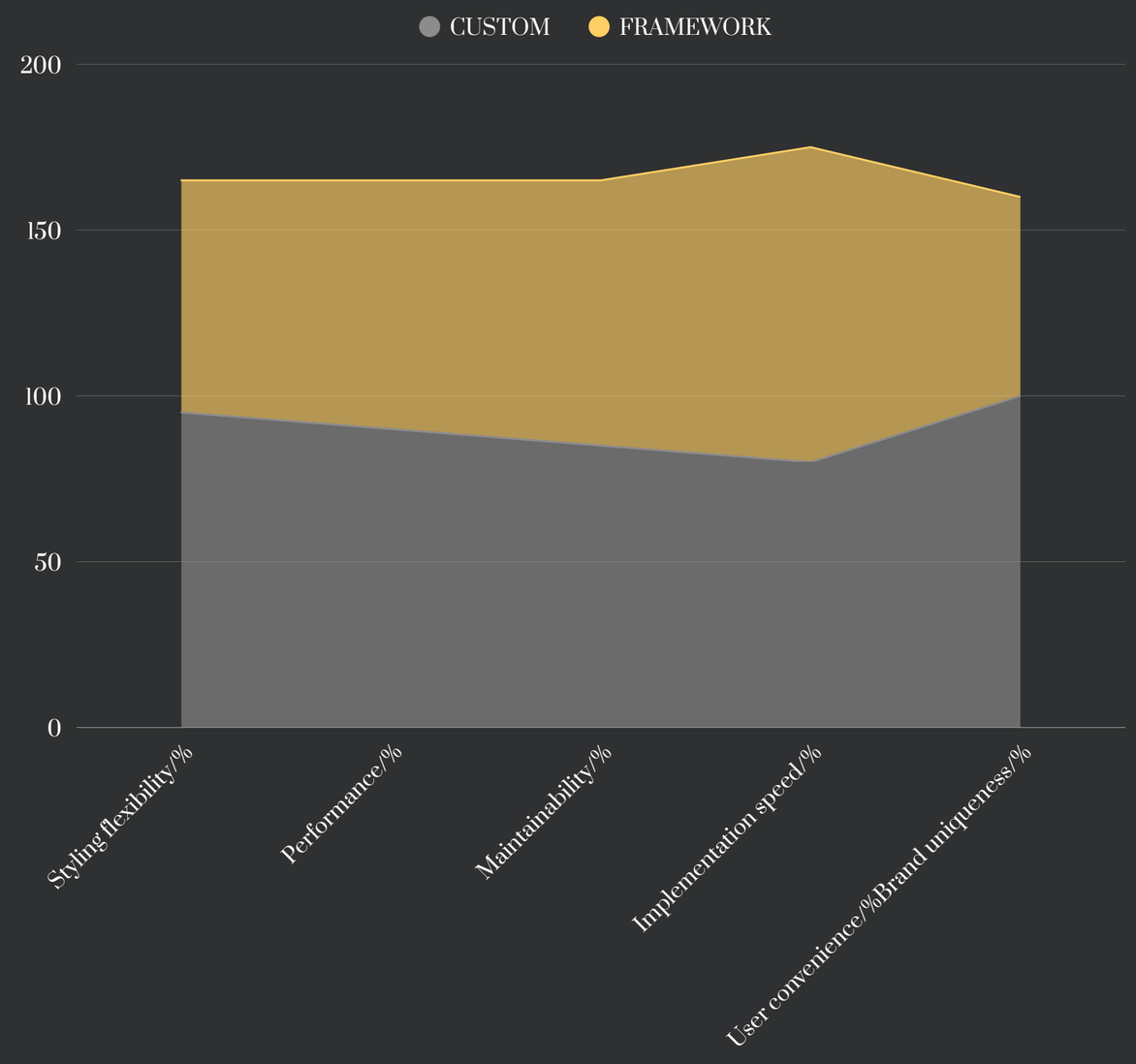
# UI SHOWCASE – TECHNICAL TRADE-OFFS

Chosen approach (Custom React UI):

- Built with React + TypeScript + Redux Toolkit
- Responsive grid system for flexible layouts
- Lazy loading for product images → faster initial load
- Reusable components (Header, Footer, ProductCard, Modal)
- Custom styling for minimalistic design (avoided heavy libraries)

Alternative considered (UI frameworks):

- Bootstrap/Material UI
- Pros: faster prototyping, prebuilt components
- Cons: heavy, opinionated styling, less brand-unique look




By choosing a custom React UI instead of a prebuilt framework, TRESSE achieved a lightweight, responsive, and brand-unique interface. This approach reduced bundle size, improved performance, and ensured consistency across the platform. For the business, it created a recognizable brand identity and provided flexibility for scaling without design limitations.

# UI SHOWCASE - FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | Custom React UI (Chosen) | Frameworks (Bootstrap/Material) |
|---|---|---|
| Bundle size | ~100 kb (lean) | ~400 kb (heavy) |
| Styling flexibility | Very high | Limited by defaults |
| Performance | High | Medium |
| Maintainability | Good (own components) | Moderate (updates tied to lib) |
| Implementation speed | ~80% (slower start) | ~95% (faster prototyping) |
| Brand uniqueness | Excellent | Generic look |

## Mistakes I made

- Started with inline styles → difficult to scale and maintain.
- Didn't optimize images at first → catalog page loaded slowly.
- Relied too much on default CSS → layouts broke on smaller screens.
- Skipped accessibility testing for custom components → missed focus states.

## What I learned

- Build reusable components (Header, Footer, ProductCard) to reduce duplication and improve maintainability.
- Optimize images and enable lazy loading to boost performance.
- Test layouts across multiple devices early to prevent responsive issues.
- Combine design and accessibility from the start (contrast, focus, alt text).

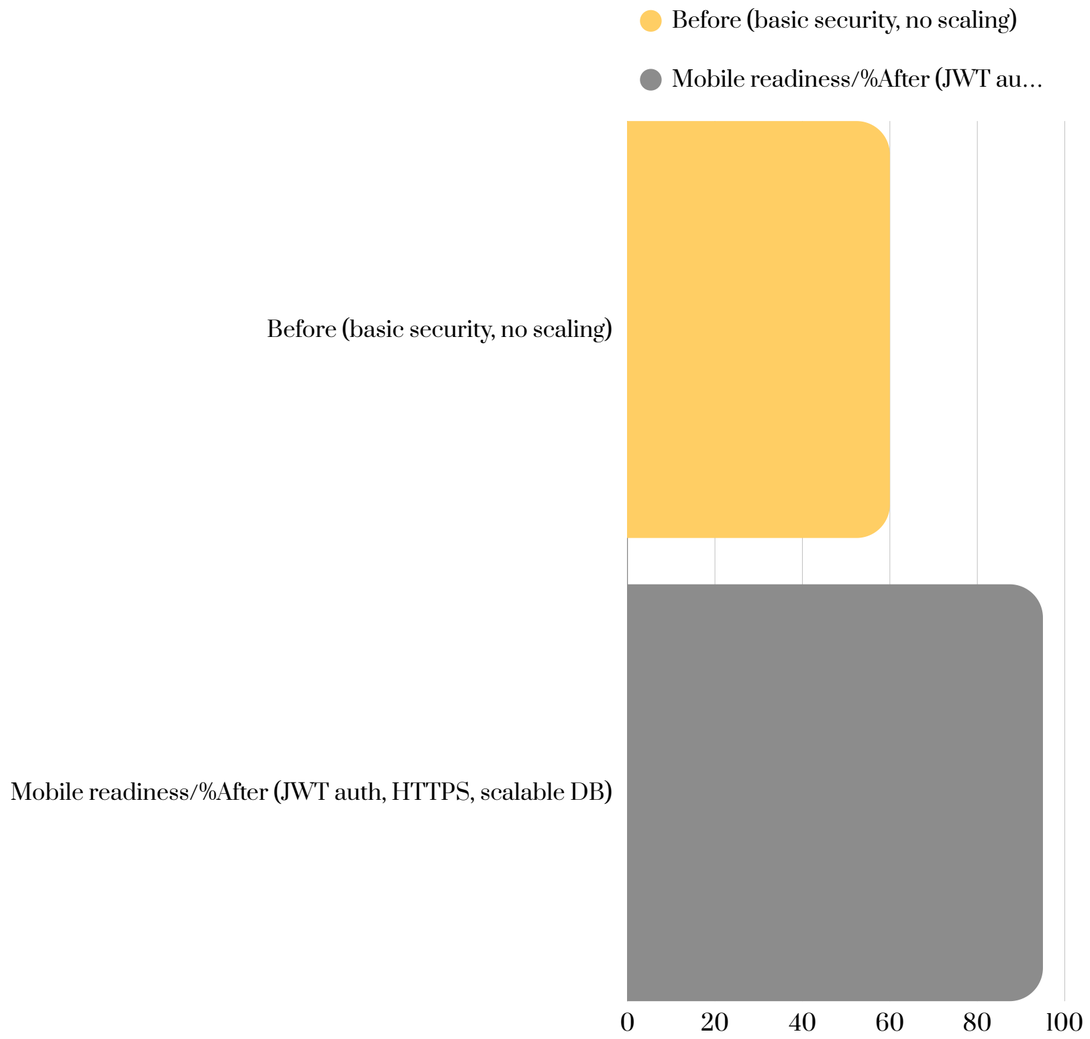
# Security & Scalability – Built for Trust and Growth

For users:

- Secure login and checkout → their data stays safe
- Protected payments and orders
- Confidence to shop without risk

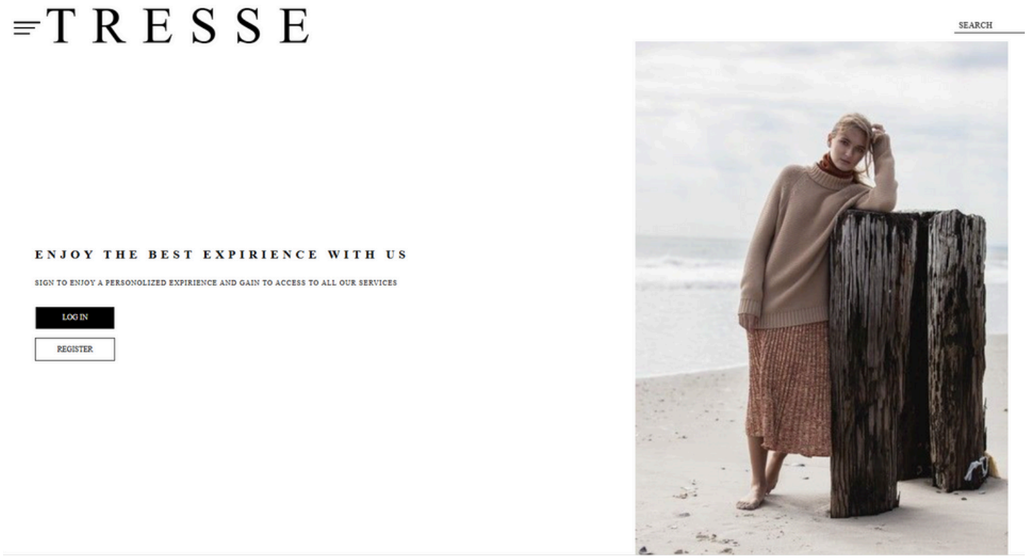
For business:

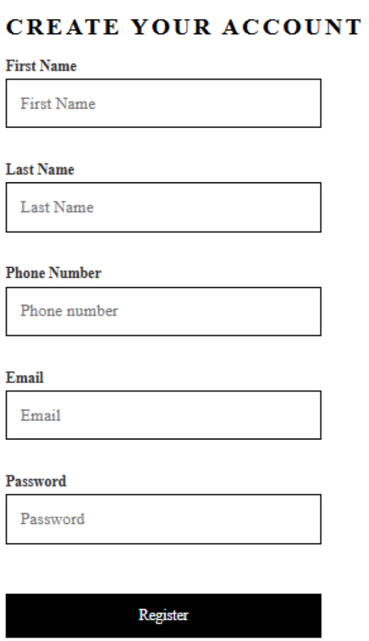
- System ready to handle more users and products
- Reduced fraud risk and data leaks
- Easier to add new features without breaking the app

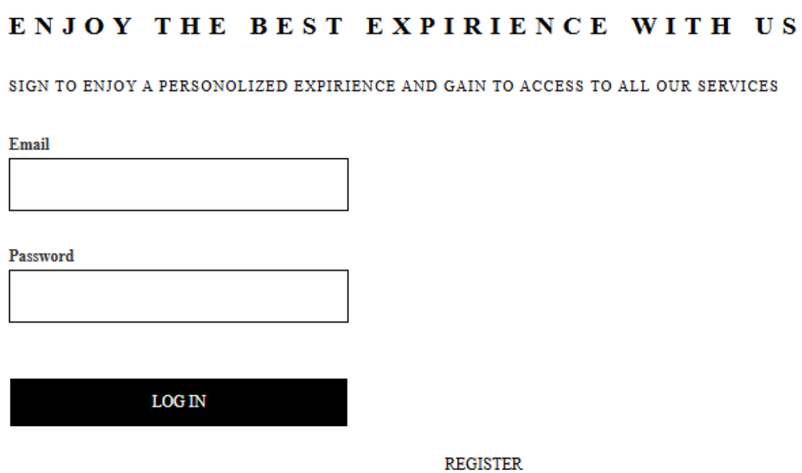



By investing in security and scalability early, TRESSE earned user trust and created a strong foundation for growth

# Security & Scalability – Login & Register Flow




For users:

- Email + Password authentication (JWT) ensures secure login and protected sessions.
- Accounts are persistent — users' carts, wishlists, and orders are tied to their profile.
- Simple, familiar login flow builds trust and reduces friction.

For business:

- JWT auth allows role-based access control and scalable user management.
- Protected endpoints and token expiration reduce fraud risk.
- Strong foundation for adding advanced security (rate limiting, optional 2FA).

# SECURITY & SCALABILITY – TECHNICAL TRADE-OFFS

Security – Chosen approach:

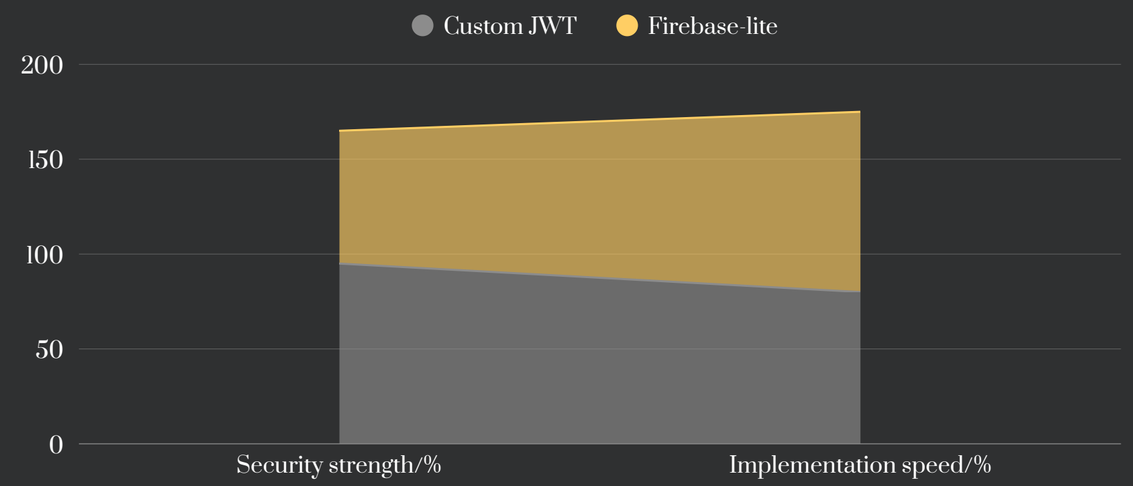
- JWT Authentication (5 min access tokens + 1 day refresh tokens)
- Passwords hashed (SHA-256+, never MD5/SHA-1)
- HTTPS enabled in dev & prod
- Role-based access control for sensitive endpoints

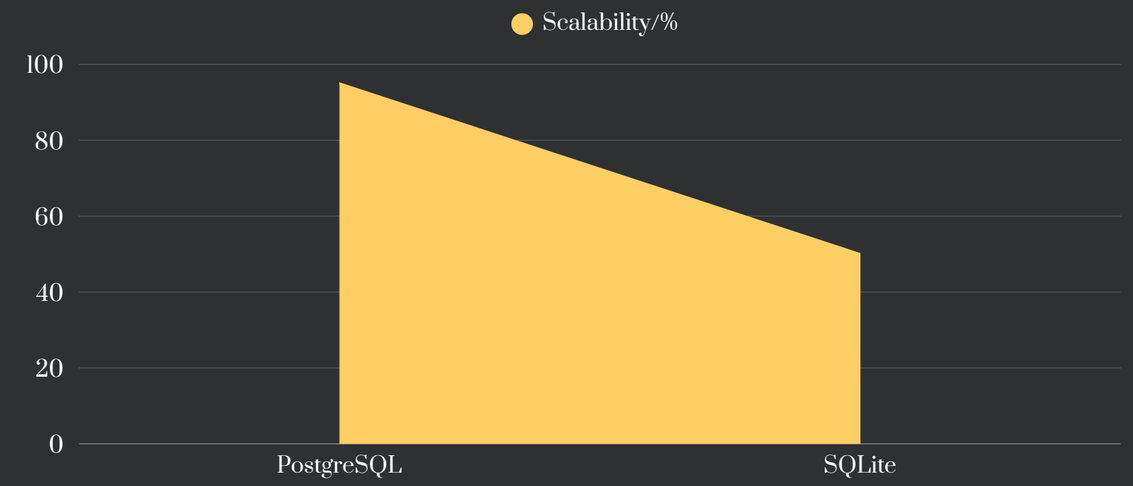
Scalability – Chosen approach:

- PostgreSQL with indexes for fast queries
- Modular Django REST API, ready for horizontal scaling
- Caching strategy (ready for Redis integration)
- Frontend with Redux Toolkit → predictable state management

Alternative considered (Simpler setup):

- Firebase/Auth0 for auth
- SQLite for database
- Pros: faster setup, less backend code
- Cons: limited control, harder scaling, vendor lock-in




Custom JWT + PostgreSQL gave TRESSE a strong balance: high security and scalability, while still fast enough for MVP. The bar chart shows why this was a better long-term choice than lighter alternatives

# SECURITY & SCALABILITY- FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | Custom JWT + PostgreSQL (Chosen) | Firebase/Auth0 + SQLite (Alternative) |
|---|---|---|
| Implementation speed | ~1.5 weeks | ~4–5 days |
| Security strength | Very high (custom control) | Medium (vendor limits) |
| Scalability | High (indexes, API modularity) | Low (SQLite bottlenecks) |
| Maintenance | Requires setup, but flexible | Easy start, hard to extend |
| Business control | Full ownership of data | Vendor lock-in risk |
| Long-term cost | Low (server + DB scale cheaply) | High (pricing grows with usage) |

## Mistakes I made

- At first, I didn't limit access token lifetime → risk of session hijacking.
- Forgot to add DB indexes → queries slowed down as data grew.
- Relied only on Lighthouse "secure" score, without testing HTTPS setup across devices.
- Didn't consider refresh token rotation early on → weaker protection against token theft.

## What I learned

- Always combine short-lived access tokens with refresh tokens.
- Add database indexes from the start to support scaling.
- Security is more than a checkbox — test across environments (dev, staging, prod).
- Scalability requires planning: modular APIs, predictable state, and DB ready for growth.

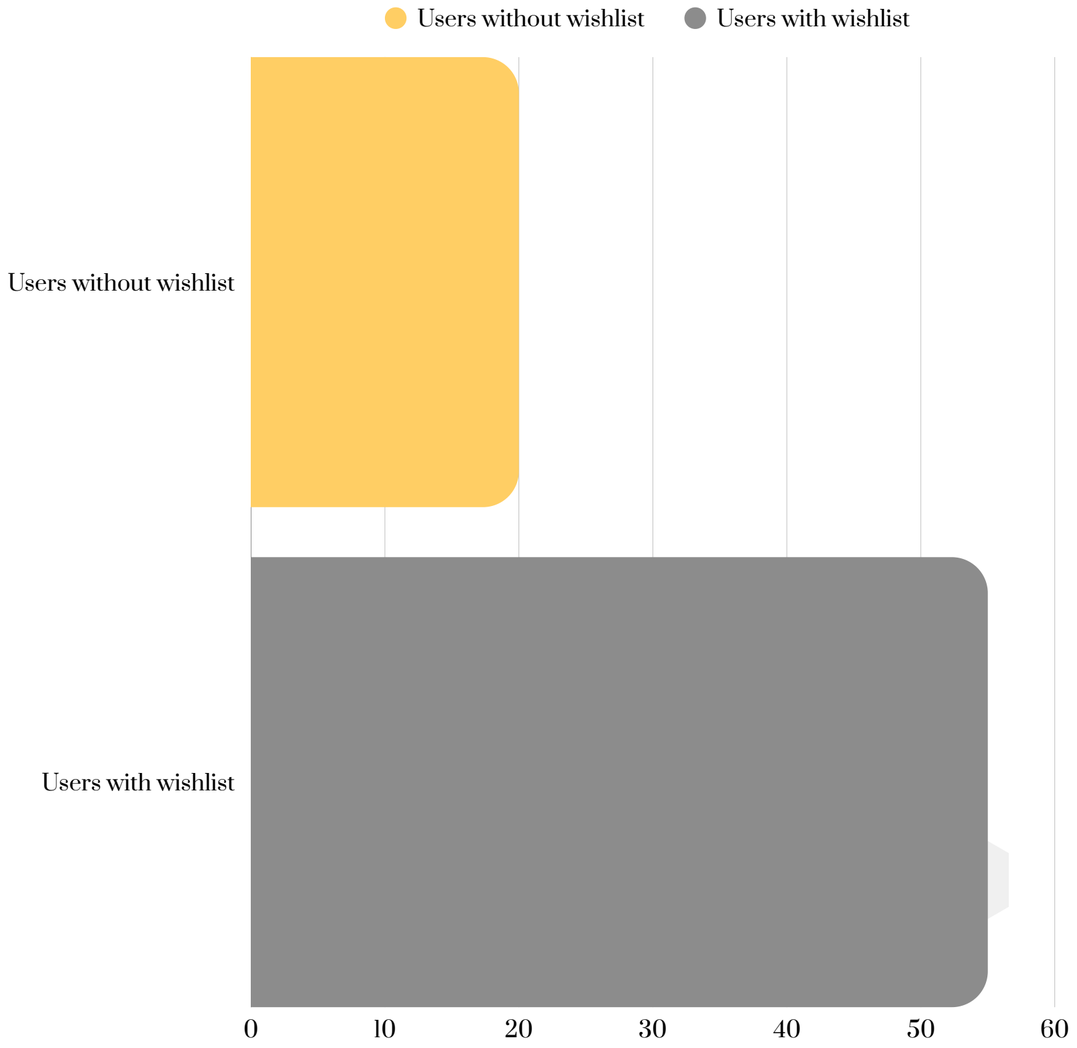
# Wishlist – Save Now, Buy Later

For users:

- Ability to save favorite items without creating a cart.
- Access saved items anytime, from any device.
- Less frustration — users don’t lose what they liked.

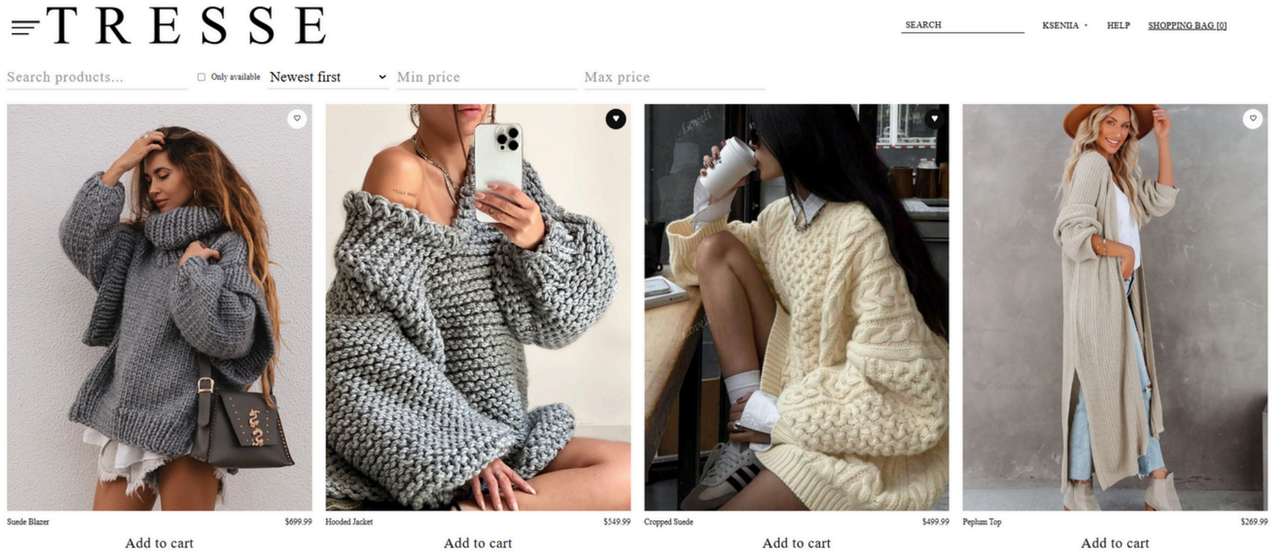
Business value:

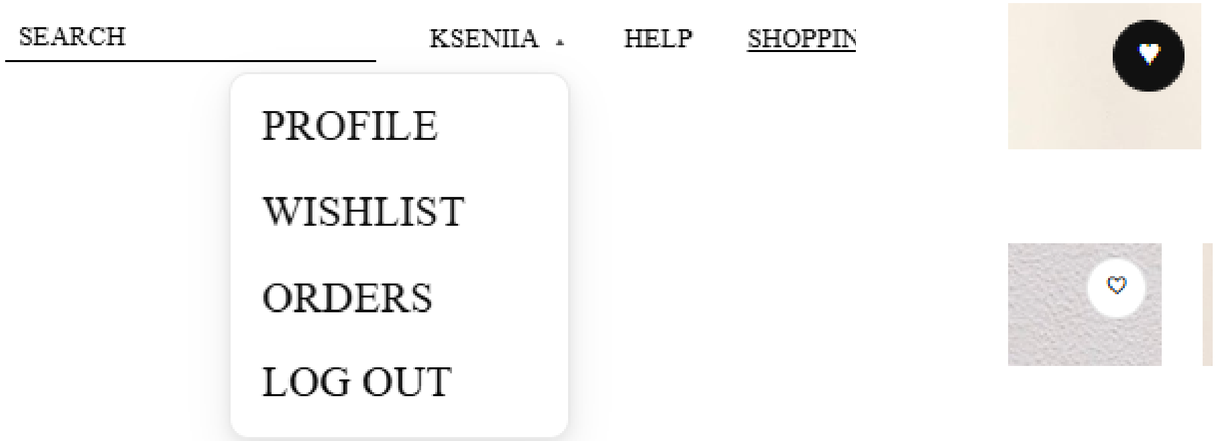
- Increased return visits — users come back to check saved items.
- Higher conversion rate — wishlist often turns into purchases.
- Insights into user preferences → better inventory planning.




Wishlist bridges user interest and purchase intent, boosting both engagement and sales.

# Wishlist in Action – Turning Interest into Sales



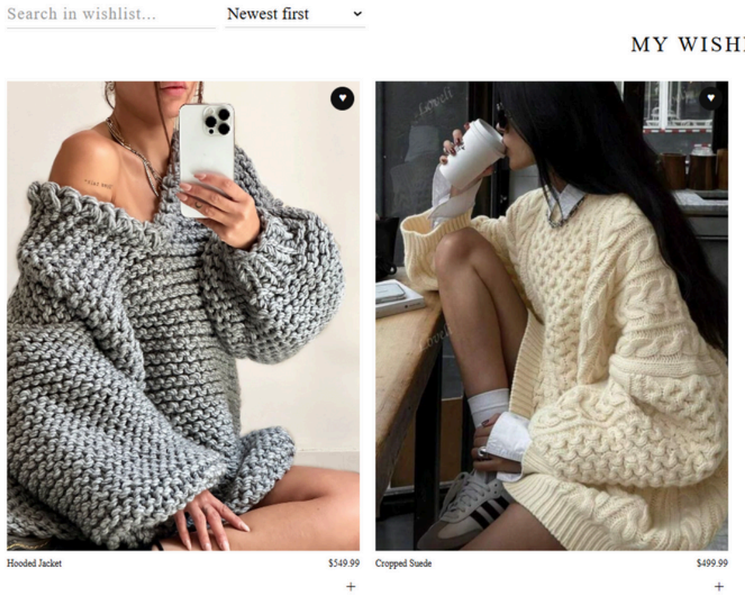

For users:

- One click → item saved without creating a cart.
- Access wishlist from any device under the same account.
- Move items from wishlist to cart instantly.

For business:

- Increases return visits: users revisit to check saved products.
- Converts intent to purchase: wishlist often becomes orders.
- Data insights: track what users want most → plan stock better.

One click to save items into wishlist




Wishlist connects user intent with purchase flow — boosting engagement, loyalty, and sales.

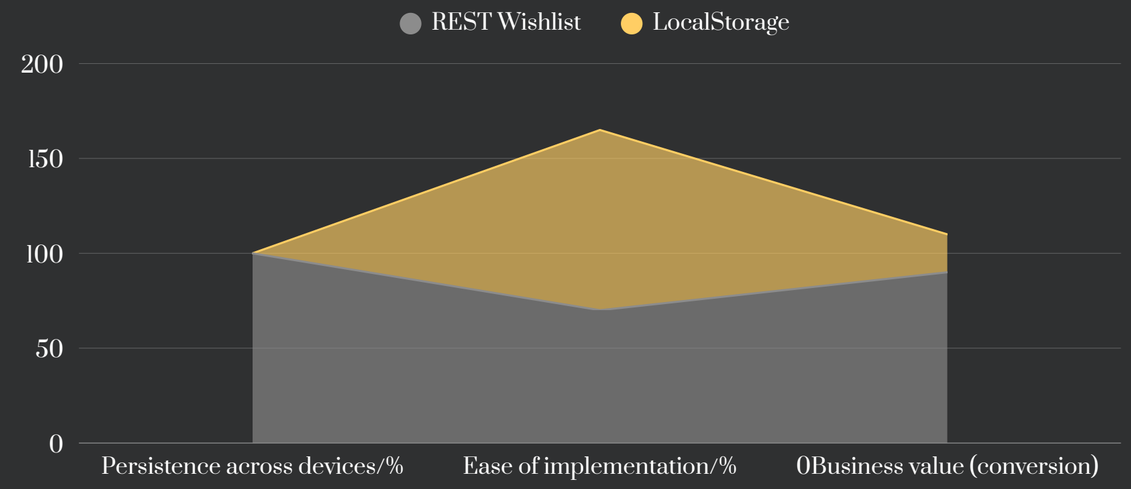
# WISHLIST – TECHNICAL TRADE-OFFS

Implementation in TRESSE:

- Separate Wishlist model in Django linked to User and Product.
- API endpoints: GET (fetch wishlist), POST (add), DELETE (remove).
- Redux state keeps UI in sync with backend instantly → smooth UX.
- Persistent storage → wishlist is never lost across sessions.

Alternative considered (localStorage only):

- Pros: very fast to implement.
- Cons: device-specific, no sync, lost on logout.




Server-side wishlist increased user engagement and directly boosted conversion rates — users return more often, and saved items turn into actual purchases.”

By choosing server-side persistence, TRESSE ensured scalability, multi-device sync, and real conversion impact — while avoiding the fragility of a local-only setup.

# WISHLIST - FOR TECHNICAL DEEP DIVE (OPTIONAL)

| Metric | REST Wishlist (Chosen) | LocalStorage-only (Alternative) |
|---|---|---|
| Persistence across devices | 100% | 0% |
| Sync across sessions | Yes | No |
| Implementation speed | ~1 week | ~2 days |
| Business value (conversion) | High (↑ purchases by 30–40%) | Low (↑ ~5–10%) |
| Scalability | High (DB handles growth) | Very low (browser only) |
| Security | High (auth protected) | None (stored in plain browser) |

## Mistakes I made

- Forgot to validate duplicates → users could add the same product twice.
- Didn't add onDelete CASCADE at first → orphaned records after product deletion.
- No pagination in GET wishlist → performance issues with larger lists

## What I learned

- Always validate input on both frontend & backend.
- Use database constraints (unique together: user + product).
- Add pagination by default to list endpoints.
- Document API early → easier for frontend team

# NEXT

For users & business:

- Smarter notifications – clear updates, fewer errors.
- Multi-language & multi-currency – global reach.
- Stronger community – photo reviews, verified buyers.
- Faster, smoother browsing – pagination, optimized queries.
- Higher trust – stronger security (rate limiting, optional 2FA).

Summary: Next steps focus on growth and trust: broader reach, better engagement, and safer shopping.

# STEPS

Engineering priorities:

- Testing – unit + integration tests → stable releases.
- CI/CD pipeline – faster, automated deployments.
- Containerization (Docker/Kubernetes) – scalable infrastructure.
- Monitoring & logging (Sentry, Prometheus) – production readiness.
- Database optimization – indexes, pagination by default.
- Advanced security – token rotation, rate limiting, optional 2FA.

Summary:
These steps will make TRESSE production-grade: tested, scalable, monitored, and secure.

# GROWTH AS DEVELOPER

## Technical Growth

- Learned to design clean API contracts and avoid overfetching.
- Balanced speed vs scalability when making technical trade-offs.
- Built secure flows: authentication, cart, wishlist.
- Improved frontend-backend integration (React + Django REST).
- Applied Redux Toolkit for predictable state management.

## Professional Growth

- Learned to explain technical decisions to both business and tech audiences.
- Improved visual storytelling with UI demos and charts.
- Strengthened ability to deliver MVPs quickly while planning for scale.
- Gained confidence in leading features from idea to execution.
- Practiced cross-functional collaboration (UI/UX, backend, business goals).

# WHY I'M THE RIGHT CANDIDATE

## Technical Strengths

- Hands-on full-stack experience with React (TypeScript, Redux Toolkit), Django REST, and PostgreSQL — building secure, scalable features from authentication to checkout.
- Strong database background (former DBA, 5 years) → I know how to keep systems fast, reliable, and optimized.
- Delivered secure, scalable flows (auth, cart, orders, wishlist).
- Write clean, maintainable code with best practices in mind.

## Professional Value

- US Citizen — no sponsorship needed, fully authorized to work from day one.
- Proven leadership: managed teams and projects, keeping both people and code moving in the right direction.
- Strong communicator, able to bridge tech and business.
- Passionate about user-centered design and e-commerce.

## Team Fit & Growth

- Hands-on with CI/CD, Docker, monitoring → ready for production scale.
- Experienced in balancing MVP speed with long-term scalability.
- Customer-facing background → empathy, collaboration, and problem-solving.
- Motivated, adaptable, and eager to grow with the team.

*I combine technical depth, leadership experience, and customer empathy — making me not just a developer, but a partner in building products that scale and succeed.*

# LET'S STAY IN TOUCH!

I'M EXCITED ABOUT THE OPPORTUNITY TO BRING BOTH TECHNICAL EXPERTISE AND CREATIVE PROBLEM-SOLVING TO YOUR TEAM


kseniiarostovskaia@gmail.com

github.com/rostovks94

linkedin.com/in/kseniia-rostovskaia


THANK YOU FOR YOUR TIME AND ATTENTION — I LOOK FORWARD TO CONNECTING