RupyaTrack - Full Stack Expense Tracker

Complete Interview Q&A; Guide

Generated: October 15, 2025

Component	Technology
Frontend	React.js + Vite, TailwindCSS, React Query, Zustand
Backend	Python Flask, Flask-JWT-Extended, SQLAlchemy
Database	PostgreSQL
Additional	Axios, @react-pdf/renderer, Lucide React

1. Project Overview

Q: Tell me about your project

A: RupyaTrack is a full-stack expense tracker application that helps users manage their personal finances efficiently.

Key Features:
- User authentication (Email/Password + OAuth with Google/GitHub)
- CRUD operations for expenses
- Trash/Restore functionality
- Real-time expense tracking with today's total
- PDF export with date filtering (Last 7/14/30 days)
- Search and filter capabilities
- Dark mode support
- Responsive design for mobile and desktop

The application uses React for the frontend with modern state management (Zustand), Flask for the backend API, and PostgreSQL for data persistence. JWT-based authentication is implemented using HTTP-only cookies for enhanced security.

2. PDF Generation

Q: How do you generate PDF of expenses?

A: I use **@react-pdf/renderer** library which creates PDFs directly in the browser using React components.

Implementation Steps:

Step 1: PDF Document Component (ExpensePdfDocument.jsx)
The component uses React-PDF primitives like Document, Page, View, Text to structure the PDF layout similar to HTML.

Step 2: PDF Download Trigger
Uses PDFDownloadLink component that generates and triggers download when clicked.

Features:
- Filter expenses by date range (7/14/30 days) before generating PDF
- Shows username, date range, total amount
- Table format with Title, Description, Amount (■), Date
- Styled with custom StyleSheet

• Client-side generation (no server required)
• Dynamic filename based on date range

3. Authentication & Security

Q: How did you implement authentication?

A: I implemented **JWT-based authentication using HTTP-only cookies** for security.

Authentication Flow:

1. Register/Login:
• User sends credentials to /auth/register or /auth/login
• Backend validates and creates JWT token
• Token stored in HTTP-only cookie (prevents XSS attacks)
• Frontend receives user data in response

2. Protected Routes:
• Backend: @jwt_required() decorator validates JWT from cookie
• Frontend: PrivateLayout checks auth state, redirects to /sign-in if not authenticated

3. Logout:
• Backend clears JWT cookie using unset_jwt_cookies()
• Frontend clears auth state

Key Security Features:
• Passwords hashed with bcrypt (via passlib)
• JWT tokens in HTTP-only cookies (not localStorage - prevents XSS)
• withCredentials: true in Axios for cookie transmission
• CORS configured to allow credentials from specific origin
• SameSite: Lax for CSRF protection

Q: Why use HTTP-only cookies instead of localStorage?

A: **Security Reasons:**
• HTTP-only cookies cannot be accessed by JavaScript → protects against XSS attacks
• localStorage can be accessed by any script → vulnerable to XSS if malicious script injected
• Cookies automatically sent with requests (no manual header management)
• SameSite attribute protects against CSRF

Trade-offs:

- Cookies require CORS configuration with credentials: true
- Requires backend to set/clear cookies
- Frontend simpler (no manual token management)

4. State Management

Q: Why did you choose Zustand over Redux?

A: **Reasons for Zustand:**
1. **Simpler API** - No boilerplate (actions, reducers, dispatch)
2. **Smaller bundle size -** ~1KB vs Redux ~10KB
3. **No Context Provider needed -** Works outside React tree
4. **TypeScript-friendly** with better type inference
5. **Performance** - Uses React's subscription model efficiently

When Redux would be better:
- Very large apps with complex state interactions
- Time-travel debugging needed
- Team already familiar with Redux ecosystem

5. Data Fetching & Caching

Q: How did you handle data fetching and why React Query?

A: I used **TanStack React Query (React Query)** for server state management.

Benefits:
1. **Automatic caching** - Reduces API calls
2. **Background refetching -** Keeps data fresh
3. **Loading/error states -** Built-in
4. **Mutations with optimistic updates**
5. **Cache invalidation** - After create/update/delete

Why not just useState + useEffect?
- Manual loading/error handling needed
- No caching (re-fetch on every mount)
- Race conditions possible
- Harder to synchronize after mutations

6. Backend Architecture

Q: Explain your Flask backend structure

A: I used **Blueprint-based modular architecture** for scalability.

Key Patterns:
1. **Application Factory Pattern** (create_app())
- Allows multiple app instances (testing, dev, prod)
- Extensions initialized with app context

2. **Blueprints for Modularity**
- /auth - Authentication routes
- /users - User management
- /expenses - Expense CRUD

3. **Configuration Classes**
- BaseConfig - Common settings
- DevConfig - Development (DEBUG=True)
- ProdConfig - Production (HTTPS, secure cookies)

Q: How did you handle database migrations?

A: Used **Flask-Migrate** (wrapper around Alembic).

Workflow:
1. Initialize migrations: flask db init (once)
2. Detect model changes: flask db migrate -m "description"
3. Apply migration: flask db upgrade
4. Rollback if needed: flask db downgrade

Benefits:
- Version control for database schema
- Rollback capability
- Automatic migration generation from model changes
- Team collaboration (migrations committed to Git)

7. Database Design

Q: Explain your database schema

A: Two main tables with relationship:

Users Table:
- id (UUID primary key)
- account_id (UUID, unique - for JWT identity)
- name, email (indexed), password_hash (nullable for OAuth)
- avatar_url, created_at

Expenses Table:
- id (UUID primary key)
- user_id (FK to users.account_id, indexed)
- title, description, amount (Numeric 12,2)
- date, expense_type, is_trashed (indexed)
- created_at, updated_at

Design Decisions:
- UUIDs instead of auto-increment IDs - Better for distributed systems
- Numeric(12,2) for amount - Avoids floating-point precision issues
- is_trashed flag - Soft delete (users can restore)
- Indexes on email, user_id, is_trashed - Query performance

Q: Why soft delete (is_trashed) instead of hard delete?

A: **Benefits:**
1. **Data recovery** - Users can restore accidentally deleted expenses
2. **Audit trail** - Track what was deleted and when
3. **Better UX** - Trash feature (like Gmail)
4. **Analytics** - Can analyze deleted items

Implementation:
- Move to trash: Set is_trashed = True
- Restore: Set is_trashed = False
- Permanent delete: Actually delete from DB (optional)
- Query: Filter by is_trashed == False for active expenses

8. API Design

Q: Explain your API response format

A: Consistent format across all endpoints:

Success Response:

```
{
  "success": true,
  "message": "Expense created successfully",
  "data": { expense object }
}
```

Error Response:

```
{
  "success": false,
  "message": "Title is required",
  "data": null
}
```

Benefits:

- Consistent error handling in frontend
- User-friendly messages for toasts/alerts
- Easy to validate - Check res.success
- Self-documenting - Clear what went wrong

Q: How did you handle CORS?

A: Configured Flask-CORS with specific settings:

Configuration:

- origins: ["http://localhost:3000"] - Specific origin (not wildcard)
- methods: GET, POST, PUT, DELETE, OPTIONS
- supports_credentials: True - Allow cookies
- allow_headers: Content-Type, Authorization

Why CORS errors happen:

- Frontend (localhost:3000) and Backend (localhost:5000) = different origins
- Browser blocks cross-origin requests by default (security)
- CORS headers tell browser "this origin is trusted"

Key Points:

- No wildcard (*) with credentials - Browser blocks this
- OPTIONS method required for preflight requests
- Frontend must use withCredentials: true in Axios

9. Frontend Architecture

Q: Explain your component structure

A: Organized by **type and feature**:

src/
- components/ - Reusable UI components
  - layouts/ (Root, Auth, Private)
  - shared/ (Logo, Loader, Sidebar)
  - widgets/ (ExpenseModal, ExpenseCard)
  - ui/ (Hero, SplashScreen)
- pages/ - Route components
  - auth/ (Sign-in, Sign-up)
  - private/ (Dashboard, Expenses, Trash)
- lib/ - Utilities and configurations
  - api.js, utils.js, validation/
- store/ - Zustand stores (auth, expense)

Benefits:
- Easy to find - Clear separation of concerns
- Reusability - Shared components
- Maintainability - Feature-specific logic in widgets
- Scalability - Add features without restructuring

Q: How did you implement form validation?

A: Used **React Hook Form + Yup** for declarative validation.

Benefits:
- Less boilerplate - No manual state management
- Schema validation - Reusable across components
- Type-safe errors - TypeScript support
- Performance - Only re-renders changed fields

Example Schema:
- title: required string
- amount: required number, positive, max 2 decimals
- date: required date
- expenseType: enum (Public/Private)

10. Performance Optimization

Q: What performance optimizations did you implement?

A: **Frontend:**
1. **React Query Caching -** Reduces API calls (5 min stale time)
2. **Lazy Loading** - Code splitting with React.lazy()
3. **Zustand -** Minimal re-renders (selective state updates)
4. **Debounced Search -** Reduces filtering operations
5. **Memoization** - React.memo() for expensive components

Backend:
1. **Database Indexes -** On email, user_id, is_trashed
2. **Query Optimization -** Only fetch required fields
3. **JWT in Cookies** - No database query per request (stateless)
4. **Connection Pooling -** SQLAlchemy manages connections

Q: How would you scale this application?

A: **Backend Scaling:**
1. **Horizontal Scaling**
- Multiple Flask instances behind load balancer (Nginx)
- Use gunicorn with multiple workers

2. **Database**
- Read replicas for queries
- Redis for caching frequent queries

3. **Async Tasks**
- Celery for PDF generation, email notifications

Frontend Scaling:
1. **CDN** - Serve static assets (Vercel, Cloudflare)
2. **Code Splitting** - Lazy load routes
3. **Service Workers** - PWA for offline capability

Database Optimization:
1. **Partitioning** - Partition expenses by date ranges
2. **Archiving -** Move old trashed items to archive table

11. Testing Strategy

Q: How would you test this application?

A: **Frontend Testing:**
1. **Unit Tests** (Vitest/Jest)

- Test utility functions (calculateTotalAmount)
- Test hooks logic

2. **Component Tests** (React Testing Library)

- Test ExpenseCard renders correctly
- Test form validation errors

3. **Integration Tests**

- Test API integration with MSW
- Test form submission flows

Backend Testing:

1. **Unit Tests** (pytest)

- Test model methods (User.to_dict())
- Test utility functions

2. **API Tests**

- Test all endpoints (status codes, response format)
- Test authentication flow

3. **Database Tests**

- Use in-memory SQLite for testing
- Test migrations up/down

12. Deployment

Q: How would you deploy this application?

A: **Backend (Flask):**

1. **Platform Options:**

- Render / Railway - Easy Python deployment
- AWS EC2 - More control
- Docker + AWS ECS - Production-grade

2. **Production Setup:**

- Use gunicorn (WSGI server): gunicorn -w 4 run:app
- Environment variables in .env
- PostgreSQL on managed service (AWS RDS, Render)

Frontend (React):

1. **Platform Options:**

- Vercel - Best for React (auto deployments)
- Netlify - Similar features
- AWS S3 + CloudFront - Custom domain

2. **Build Process:**

- npm run build - Creates dist/ folder

- Upload to hosting platform

CI/CD:
- GitHub Actions workflow
- Run tests before deploy
- Auto-deploy on merge to main

13. Security Best Practices

Q: What security measures did you implement?

A: 1. **Password Security**
- Bcrypt hashing (not plain text)
- Min 8 characters enforced

2. **JWT Security**
- HTTP-only cookies (prevents XSS)
- SameSite attribute (prevents CSRF)
- 7-day expiry
- Secure flag in production (HTTPS only)

3. **Input Validation**
- Backend: Validate all inputs
- Frontend: Yup schemas
- Prevent SQL injection (SQLAlchemy ORM)

4. **CORS**
- Whitelist specific origin (no wildcard)
- Only required headers allowed

5. **Environment Variables**
- Secrets in .env (not committed)
- Different secrets for dev/prod

14. Challenges & Problem-Solving

Q: What was the biggest challenge you faced?

A: **Challenge: CORS + Cookie Authentication**

Problem:
- Frontend couldn't send/receive cookies from backend
- 308 redirect errors on OPTIONS preflight
- Cookies not persisted after login

Solution:
1. **Backend:**
- Fixed route trailing slashes (@bp.route("") not "/")
- CORS config with supports_credentials=True
- JWT_TOKEN_LOCATION = ["cookies"]
2. **Frontend:**
- Added withCredentials: true in Axios
3. **Debugging:**
- Checked Network tab for cookies
- Added console.logs in middleware
- Tested with curl to isolate issue

Lesson Learned:
- Test cross-origin scenarios early
- Understand browser security policies
- Use DevTools Network tab extensively

15. Future Enhancements

Q: What features would you add next?

A: **High Priority:**
1. **Categories/Tags**
- Group expenses (Food, Transport)
- Filter by category, pie charts
2. **Budget Tracking**
- Set monthly budget
- Alerts when approaching limit
3. **Recurring Expenses**
- Auto-create monthly subscriptions

Medium Priority:
4. **Data Visualization**
- Chart.js for spending trends
- Monthly comparison graphs

5. **Notifications**
• Email summary (weekly/monthly)
• Push notifications for budget
6. **Export Formats**
• CSV export, Excel with formulas

Low Priority:
7. **Mobile App** - React Native
8. **Collaborative Budgets** - Share with family
9. **AI Insights** - Spending pattern analysis

16. Code Quality

Q: How do you ensure code quality?

A: **Frontend:**
• ESLint - Catch errors and enforce style
• Prettier - Consistent formatting
• Prop Types or TypeScript
• Code Reviews
• Component Reusability - DRY principle

Backend:
• Flake8 - Python linting
• Black - Python formatter
• Type Hints for function signatures
• Docstrings for complex functions
• Error Handling - Try-except blocks

Git Workflow:
• Feature branches from develop
• Pull requests with reviews
• Conventional commits (feat:, fix:, refactor:)

Documentation:
• README with setup instructions
• API documentation (Swagger/OpenAPI)
• Code comments for complex logic

17. Additional Technical Questions

Q: Explain controlled vs uncontrolled components

A: **Controlled Component:**
• React state is source of truth
• Every change triggers state update
• Example: <input value={value} onChange={setValue} />

Uncontrolled Component:
• DOM is source of truth
• Use refs to access value
• Good for file uploads

In RupyaTrack:
• Used controlled via React Hook Form
• Benefits: Validation on change, conditional rendering

Q: What is the Virtual DOM and why is it fast?

A: **Virtual DOM:**
• Lightweight JavaScript representation of real DOM
• React keeps virtual DOM in memory

Why Fast:
1. **Batch Updates** - Multiple changes grouped
2. **Diffing Algorithm** - Only changed elements updated
3. **Reconciliation** - Minimal changes to real DOM

Example in RupyaTrack:
• List of 100 expenses
• Adding 1 expense doesn't re-render all 100
• Only new ExpenseCard rendered

Q: Explain async/await vs Promises

A: **Promises:**
getExpenses().then(expenses => setExpenses(expenses))

Async/Await:
const expenses = await getExpenses();
setExpenses(expenses);

Why Async/Await Preferred:

- Reads like synchronous code
- Better error handling with try-catch
- Avoid callback hell

In RupyaTrack:
Used async/await in all API calls for cleaner code

18. Interview Tips

Q: How to present your project effectively?

A: **1. Start with Overview:**
- "RupyaTrack is a full-stack expense tracker..."
- Mention tech stack upfront
- Highlight 2-3 key features

2. Show Enthusiasm:
- Discuss challenges you enjoyed solving
- Mention what you learned

3. Be Honest:
- If you don't know, say so
- Explain how you'd research it

4. Prepare Demo:
- Have app running locally
- Show user flow (register → create → PDF)
- Point out technical aspects

5. Know Your Code:
- Explain any file they ask about
- Walk through API call flow
- Explain decision-making process