Sprint 1: Core Foundation - JIRA Epic & Stories

EPIC: CRYPTO-EPIC-001 - Authentication & Core Infrastructure

Epic Name: User Authentication and Security Foundation

Epic Summary: Establish secure user authentication system with 2FA, session management, and foundational infrastructure for the cryptocurrency portfolio tracker.

Epic Description: Implement comprehensive authentication system including user registration with email verification, multi-factor authentication using TOTP, secure session management with JWT tokens, and foundational security controls. This epic establishes the security-first approach required for handling sensitive financial data and API keys in cryptocurrency portfolio management.

Business Value: Without secure authentication, users cannot safely access the platform or connect their exchange accounts. This is the foundational requirement for all subsequent features.

Acceptance Criteria:

- Users can register with email verification
- 2FA is enforced for all accounts
- Sessions are managed securely with automatic timeout
- All authentication endpoints pass security testing
- Audit logging captures all authentication events

Tech Stack: React 18 + TypeScript (Frontend), Python 3.11+ (Backend) using FastAPI + Pydantic + Async SQLAlchemy, PostgreSQL (Database), Redis (Session cache)

Story Points: 34

Target Release: Sprint 1 (Oct 1-15, 2025)

Linked Requirements: CRYPTO-F-001, CRYPTO-F-002, CRYPTO-F-003, CRYPTO-SR-001, CRYPTO-SR-003, CRYPTO-SR-004, CRYPTO-SR-005

EPIC: CRYPTO-EPIC-002 - Portfolio Management Core

Epic Name: Basic Portfolio Creation and Management

Epic Summary: Enable users to create multiple portfolios and manually add cryptocurrency holdings with real-time price tracking.

Epic Description: Build the core portfolio management system allowing users to organize their cryptocurrency investments across multiple portfolios. Users can manually input holdings with purchase details, view real-time prices via WebSocket connections, and see an aggregated dashboard of their total portfolio value. This epic provides the essential functionality for users to track their crypto investments.

Business Value: This is the primary value proposition of the application - users can track and manage their cryptocurrency portfolios in one place.

Acceptance Criteria:

- Users can create and manage multiple portfolios
- Manual holdings can be added with all required fields
- Real-time prices display with <10s latency

- Dashboard shows accurate portfolio valuations
- All portfolio operations are tested and validated

Tech Stack: React 18 + TypeScript, Python 3.11+ (FastAPI backend), Async SQLAlchemy 2.0, PostgreSQL, Redis, FastAPI WebSockets + aiohttp, Market Data APIs (CoinGecko/CoinMarketCap)

Backend migration: Rust → Python component mapping

Old (Rust)	New (Python)
axum handlers / routes	FastAPI APIRouters (async endpoints)
Rust structs / SeaORM models	SQLAlchemy 2.0 async models (Declarative)
tokio background jobs	Celery tasks (async where possible)
totp-rs crate	pyotp + qrcode
jsonwebtoken crate	PyJWT
aes_gcm crate	cryptography (AES-GCM)
Redis client (rust)	aioredis / redis-py (async)
WebSocket server (axum)	FastAPI WebSocket endpoints
Rust unit tests (cargo)	pytest + httpx (async)

The rest of the document preserves acceptance criteria, database schemas, security requirements and frontend stack.

Story Points: 42

Target Release: Sprint 1 (Oct 1-15, 2025)

Linked Requirements: CRYPTO-F-004, CRYPTO-F-005, CRYPTO-F-007, CRYPTO-F-010, CRYPTO-NF-001

USER STORIES (JIRA Format)

Story 1: User Registration with Email Verification

Story ID: CRYPTO-US-001

Story Name: User Registration with Email Verification

Story Type: Story

Epic Link: CRYPTO-EPIC-001

Priority: Highest

Story Points: 5

Sprint: Sprint 1

Assignee: Frontend + Backend Team

Reporter: Product Owner

Labels: authentication, security, frontend, backend

Summary: As a new user, I want to register for an account with my email and password so that I can securely access the cryptocurrency portfolio tracker.

Description: Implement user registration flow with email verification to ensure secure account creation. The system must validate email format and password strength in real-time, send verification emails, and prevent duplicate registrations.

User Story (Narrative):

```
AS A new user
I WANT TO register with my email and create a secure password
SO THAT I can access the platform and start tracking my cryptocurrency portfolio
```

Acceptance Criteria:

```
GIVEN I am on the registration page
WHEN I enter valid email and password (min 8 chars, 1 special char, 1 number)
THEN my account is created and verification email is sent

GIVEN I have registered but not verified
WHEN I try to log in
THEN I am prompted to verify my email first

GIVEN I click the verification link in email
WHEN the token is valid and not expired (24hr window)
THEN my account is activated and I can log in

GIVEN I try to register with existing email
WHEN I submit the form
THEN I see error message "Email already registered"

GIVEN I enter invalid email format
WHEN I type in the email field
THEN I see real-time validation error below field
```

Technical Implementation Details:

Frontend (React + TypeScript):

- Registration form component with Formik validation
- Real-time password strength indicator
- Email format validation using regex
- Error message display components
- Success confirmation page

Backend (Python — FastAPI + Pydantic + passlib):

- `/api/auth/register` POST endpoint (async)
- Email validation logic using Pydantic validators

- Password hashing using passlib (bcrypt) via async-friendly helpers
- Email verification token generation using PyJWT (or itsdangerous) with 24h expiry
- Async SQLAlchemy 2.0 for DB transactions (async_sessionmaker)
- SendGrid (or an async SMTP client) integration for email delivery

Example: async FastAPI register endpoint (simplified)

```
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.ext.asyncio import AsyncSession
from passlib.context import CryptContext
from pydantic import BaseModel, EmailStr
import jwt

pwd_ctx = CryptContext(schemes=["bcrypt"], deprecated="auto")
router = APIRouter()

class UserCreate(BaseModel):
  email: EmailStr
  password: str

@router.post("/api/auth/register", status_code=201)
async def register(data: UserCreate, db: AsyncSession = Depends(get_async_db)):
  hashed = pwd_ctx.hash(data.password)
  # create user with async SQLAlchemy, send verification email with JWT token
  # omitted: duplicate email check, transaction handling, SendGrid call
  return {"message": "verification email sent"}
```

SQLAlchemy/ Pydantic models are used end-to-end for validation and persistence.

Database (PostgreSQL):

```
CREATE TABLE users (
  id UUID PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  email_verified BOOLEAN DEFAULT FALSE,
  verification_token VARCHAR(255),
  verification_expires_at TIMESTAMP,
  created_at TIMESTAMP DEFAULT NOW()
);
```

Testing Requirements:

- Unit tests: Email validation, password strength check (Jest)
- Unit tests: Password hashing, token generation (Cargo test)
- Integration test: Full registration flow (Cypress)
- API test: Registration endpoint validation (Postman)
- Security test: Input sanitization, SQL injection prevention (OWASP ZAP)

Definition of Done:

- [ ] Frontend registration form completed and reviewed

- [ ] Backend registration endpoint implemented
- [ ] Email service integrated and tested
- [ ] Database schema created and migrated
- [ ] Unit tests written and passing (95%+ coverage)
- [ ] Integration tests passing
- [ ] API tests documented in Postman
- [ ] Security review completed
- [ ] Code merged to develop branch
- [ ] Test cases: TC-Auth-01, TC-Auth-02, TC-Auth-03 passed

Linked Requirements: CRYPTO-F-001, CRYPTO-SR-005

Test Cases: TC-Auth-01, TC-Auth-02, TC-Auth-03

Components:

- Frontend: `RegisterForm.tsx`, `EmailVerification.tsx`
- Backend: `auth_service.rs`, `email_service.rs`
- Database: `users` table

Dependencies:

- SendGrid API credentials configured
- Email templates designed and approved
- Database schema approved

Story 2: Multi-Factor Authentication (2FA) Implementation

Story ID: CRYPTO-US-002

Story Name: TOTP-Based 2FA Implementation

Story Type: Story

Epic Link: CRYPTO-EPIC-001

Priority: Highest

Story Points: 8

Sprint: Sprint 1

Assignee: Backend + Security Team

Reporter: Security Lead

Labels: authentication, security, 2FA, backend, frontend

Summary: As a security-conscious user, I want to enable 2FA on my account so that my cryptocurrency portfolio is protected even if my password is compromised.

Description: Implement TOTP-based two-factor authentication supporting authenticator apps (Google Authenticator, Authy). System generates QR codes, validates TOTP tokens, provides backup codes, and enforces 2FA during login with rate limiting.

User Story (Narrative):

```
AS A security-conscious user
I WANT TO enable two-factor authentication using an authenticator app
SO THAT my account remains secure even if my password is stolen
```

Acceptance Criteria:

```
GIVEN I am logged in to my account
WHEN I navigate to security settings and click "Enable 2FA"
THEN I see a QR code and setup instructions

GIVEN I scan the QR code with Google Authenticator
WHEN I enter the 6-digit code shown in the app
THEN my 2FA is activated and I receive 10 backup codes

GIVEN 2FA is enabled on my account
WHEN I log in with email and password
THEN I am prompted for 2FA code before access is granted

GIVEN I enter invalid 2FA code 5 times
WHEN I try the 6th time
THEN my account is locked for 15 minutes

GIVEN I have lost access to my authenticator app
WHEN I use one of my backup codes
THEN I can log in successfully and that backup code is invalidated

GIVEN 2FA is enabled
WHEN I attempt to disable it
THEN I must enter current password and 2FA code for confirmation
```

Technical Implementation Details:

Frontend (React + TypeScript):

- 2FA setup modal with QR code display
- TOTP code input component (6-digit field)
- Backup codes display and download functionality
- 2FA verification during login flow
- Security settings management UI

Backend (Python — FastAPI + pyotp + qrcode):

- TOTP using `pyotp` for generation & verification
- `/api/auth/2fa/setup` POST endpoint (returns provisioning URI / QR code)
- `/api/auth/2fa/verify` POST endpoint
- `/api/auth/2fa/backup-codes` POST endpoint (generate and hash backup codes)
- Secrets stored encrypted using `cryptography` (AES-GCM) with a KMS/master key
- Rate limiting via Redis (e.g., sliding-window or leaky-bucket) — max 5 attempts per 15 min

Example: generate TOTP provisioning URI

```
import pyotp
from qrcode import make as qrcode_make

secret = pyotp.random_base32()
totp = pyotp.TOTP(secret)
uri = totp.provisioning_uri(name="user@example.com", issuer_name="DeFiers")
# Optionally create a QR image and return it (as SVG/PNG)
```

Database (PostgreSQL):

```
ALTER TABLE users ADD COLUMN totp_secret VARCHAR(255);
ALTER TABLE users ADD COLUMN totp_enabled BOOLEAN DEFAULT FALSE;

CREATE TABLE backup_codes (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  code_hash VARCHAR(255) NOT NULL,
  used BOOLEAN DEFAULT FALSE,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE login_attempts (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  ip_address VARCHAR(45),
  attempt_type VARCHAR(50),
  success BOOLEAN,
  attempted_at TIMESTAMP DEFAULT NOW()
);
```

Testing Requirements:

- Unit tests: TOTP generation and validation (Cargo test)
- Unit tests: Backup code generation and hashing (Cargo test)
- Component tests: 2FA setup flow (Jest + React Testing Library)
- Integration test: Full 2FA enablement and login (Cypress)
- API tests: 2FA endpoints (Postman collection)
- Security test: Rate limiting validation (JMeter)
- Security test: Backup code security (OWASP ZAP)
- Penetration test: Brute force attack prevention

Definition of Done:

- [ ] TOTP library integrated and configured
- [ ] Frontend 2FA UI components completed
- [ ] Backend 2FA endpoints implemented
- [ ] QR code generation working correctly
- [ ] Backup codes securely generated and stored
- [ ] Rate limiting middleware deployed
- [ ] Database schema updated

- [ ] Unit tests passing (95%+ coverage)
- [ ] Integration tests passing
- [ ] Security testing completed and approved
- [ ] Code review completed
- [ ] Documentation updated
- [ ] Test cases: TC-Auth-04, TC-Auth-05, TC-Auth-06 passed

Linked Requirements: CRYPTO-F-002, CRYPTO-SR-001, CRYPTO-SR-002

Test Cases: TC-Auth-04, TC-Auth-05, TC-Auth-06, TC-Sec-02, TC-Sec-03

Components:

- Frontend: `TwoFactorSetup.tsx`, `TwoFactorVerify.tsx`, `BackupCodes.tsx`
- Backend: `totp_service.rs`, `rate_limiter.rs`
- Database: `totp_secret` column, `backup_codes` table, `login_attempts` table

Dependencies:

- `totp-rs` crate installed
- QR code generation library configured
- Rate limiting infrastructure ready

Story 3: Secure Session Management with JWT

Story ID: CRYPTO-US-003

Story Name: JWT-Based Session Management and Timeout

Story Type: Story

Epic Link: CRYPTO-EPIC-001

Priority: Highest

Story Points: 8

Sprint: Sprint 1

Assignee: Backend + Frontend Team

Reporter: Security Lead

Labels: authentication, security, session-management, backend, frontend

---

Summary: As a returning user, I want to log in securely with automatic session timeout so that my account remains protected when I'm inactive.

Description: Implement secure login flow with JWT token-based session management. Sessions expire after 30 minutes of inactivity or 24 hours maximum. Failed login attempts are logged and rate-limited. Users can view and manage active sessions across devices.

User Story (Narrative):

```
AS A returning user
I WANT TO log in securely with automatic session timeout
```

```
SO THAT my account is protected if I forget to log out
```

Acceptance Criteria:

```
GIVEN I have a verified account with 2FA enabled
WHEN I enter correct email, password, and 2FA code
THEN I am logged in and JWT token is issued

GIVEN I am logged in and inactive for 30 minutes
WHEN I try to access a protected page
THEN I am automatically logged out and redirected to login

GIVEN my session has been active for 24 hours
WHEN I try to make any request
THEN my session expires regardless of activity

GIVEN I fail to log in 5 times
WHEN I try the 6th attempt
THEN my account is temporarily locked for 15 minutes

GIVEN I am logged in on multiple devices
WHEN I view my security settings
THEN I can see all active sessions with device info and location

GIVEN I am logged in on Device A
WHEN I remotely log out from Device B
THEN Device A session is immediately terminated
```

Technical Implementation Details:

Frontend (React + TypeScript):

- Login form with email, password, 2FA code inputs
- JWT token storage in httpOnly cookies
- Axios interceptors for token refresh
- Automatic redirect on 401 responses
- Session timeout warning modal (5 min before expiry)
- Active sessions management page

Backend (Python — FastAPI + PyJWT + Redis):

- `/api/auth/login` POST endpoint (async) validates password and 2FA if enabled
- Issue access/refresh tokens via PyJWT with proper claims and short TTLs
- Store session metadata in Redis (token hash -> user_id, last_activity, expires_at)
- `/api/auth/logout` POST endpoint (revoke session in Redis and DB)
- `/api/auth/sessions` GET endpoint (list sessions backed by Redis and DB)
- `/api/auth/sessions/{id}` DELETE endpoint (revoke session)
- Rate limiting enforcement via Redis
- Audit logging written to `security_events` table asynchronously

Notes: recommend storing refresh tokens server-side (Redis) to support revocation and last_activity updates.

Database (PostgreSQL + Redis):

PostgreSQL:

```
CREATE TABLE sessions (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  token_hash VARCHAR(255) NOT NULL,
  device_info JSONB,
  ip_address VARCHAR(45),
  last_activity TIMESTAMP,
  expires_at TIMESTAMP,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_sessions_user_id ON sessions(user_id);
CREATE INDEX idx_sessions_expires_at ON sessions(expires_at);
```

Redis:

```
Key: session:{token_hash}
Value: {user_id, expires_at, last_activity}
TTL: 24 hours
```

Testing Requirements:

- Unit tests: JWT generation and validation (Cargo test)
- Unit tests: Session timeout logic (Cargo test)
- Component tests: Login form validation (Jest)
- Integration test: Full login and logout flow (Cypress)
- Integration test: Session timeout behavior (Cypress)
- API tests: Authentication endpoints (Postman)
- Load test: Concurrent login requests (JMeter)
- Security test: JWT token security (OWASP ZAP)
- Security test: Session hijacking prevention

Definition of Done:

- [ ] JWT implementation completed with signing key rotation
- [ ] Login endpoint with 2FA integration working
- [ ] Session timeout mechanisms implemented
- [ ] Redis session storage configured
- [ ] Frontend session management implemented
- [ ] Rate limiting deployed and tested
- [ ] Audit logging capturing all auth events
- [ ] Unit tests passing (95%+ coverage)
- [ ] Integration tests passing
- [ ] Security testing completed
- [ ] Load testing with 1000+ concurrent sessions passed
- [ ] Documentation updated
- [ ] Test cases: TC-Auth-07, TC-Auth-08, TC-Sec-04 passed

Linked Requirements: CRYPTO-F-003, CRYPTO-SR-003, CRYPTO-SR-004

Test Cases: TC-Auth-07, TC-Auth-08, TC-Sec-04, TC-Sec-05

Components:

- Frontend: `LoginForm.tsx`, `SessionManager.tsx`, `SessionTimeout.tsx`
- Backend: `jwt_service.rs`, `session_middleware.rs`, `audit_logger.rs`
- Database: `sessions` table, Redis session cache

Dependencies:

- JWT library (`jsonwebtoken` crate)
- Redis connection configured
- Audit logging infrastructure ready

Story 4: Portfolio Creation and Management

Story ID: CRYPTO-US-004

Story Name: Create and Manage Multiple Portfolios

Story Type: Story

Epic Link: CRYPTO-EPIC-002

Priority: High

Story Points: 5

Sprint: Sprint 1

Assignee: Full Stack Team

Reporter: Product Owner

Labels: portfolio, crud, frontend, backend

Summary: As an investor with different investment strategies, I want to create and manage multiple portfolios so that I can organize my holdings by purpose (e.g., long-term, trading, DeFi).

Description: Build portfolio CRUD operations allowing users to create multiple portfolios with custom names and descriptions. Each portfolio is isolated for calculations and can be set as default for quick access.

User Story (Narrative):

```
AS AN investor with multiple investment strategies
I WANT TO create separate portfolios for each strategy
SO THAT I can organize and track my holdings independently
```

Acceptance Criteria:

```
GIVEN I am logged in
WHEN I navigate to portfolios page and click "Create Portfolio"
THEN I can enter name (required, max 100 chars) and description (optional, max 500 chars)
```

```
GIVEN I have created a portfolio
WHEN I view my portfolios list
THEN I see portfolio name, description, total value, asset count, and creation date

GIVEN I have multiple portfolios
WHEN I click on any portfolio
THEN I see detailed view with all holdings in that portfolio

GIVEN I want to update a portfolio
WHEN I click edit and change name or description
THEN changes are saved and reflected immediately

GIVEN I want to delete a portfolio
WHEN I click delete and confirm the warning
THEN portfolio and all associated holdings are permanently removed

GIVEN I have multiple portfolios
WHEN I set one as "default"
THEN it appears first in list and on dashboard after login

GIVEN portfolios are device-synced
WHEN I create/edit portfolio on web
THEN changes appear on mobile app immediately
```

Technical Implementation Details:

Frontend (React + TypeScript):

- Portfolio list component with cards
- Portfolio creation modal/form
- Portfolio edit inline or modal form
- Portfolio deletion confirmation dialog
- Default portfolio toggle/badge
- Empty state for new users
- Loading skeletons during data fetch

Backend (Python — FastAPI + Async SQLAlchemy 2.0):

- Async SQLAlchemy declarative models for `Portfolio` and relations
- CRUD endpoints mounted on APIRouter under `/api/portfolios` using async DB sessions
- Validation using Pydantic request/response schemas
- Soft-delete / cascade behaviors implemented at DB level and in SQLAlchemy models
- Use transactions (async) for create/update/delete to ensure consistency

Example SQLAlchemy 2.0 async model (simplified):

```
from sqlalchemy import Column, String, Boolean, ForeignKey
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.orm import relationship, declarative_base

Base = declarative_base()

class Portfolio(Base):
```

```
    __tablename__ = 'portfolios'
    id = Column(UUID(as_uuid=True), primary_key=True)
    user_id = Column(UUID(as_uuid=True), ForeignKey('users.id', ondelete='CASCADE'))
    name = Column(String(100), nullable=False)
    description = Column(String)
    is_default = Column(Boolean, default=False)
```

Use `async_sessionmaker` and dependency injection to provide `AsyncSession` in endpoints.

Database (PostgreSQL):

```
CREATE TABLE portfolios (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  name VARCHAR(100) NOT NULL,
  description TEXT,
  is_default BOOLEAN DEFAULT FALSE,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_portfolios_user_id ON portfolios(user_id);
CREATE UNIQUE INDEX idx_portfolios_user_default ON portfolios(user_id) WHERE is_default =
TRUE;
```

Testing Requirements:

- Unit tests: Portfolio validation logic (Cargo test)
- Component tests: Portfolio form validation (Jest)
- Component tests: Portfolio list rendering (Jest + React Testing Library)
- Integration test: Full portfolio CRUD flow (Cypress)
- API tests: Portfolio endpoints (Postman collection)
- Test: Cross-portfolio data isolation
- Test: Default portfolio enforcement (only one per user)

Definition of Done:

- [ ] Frontend portfolio components completed
- [ ] Backend portfolio CRUD endpoints implemented
- [ ] Database schema created with constraints
- [ ] Validation rules enforced (client and server)
- [ ] Unit tests passing (95%+ coverage)
- [ ] Component tests passing
- [ ] Integration tests passing
- [ ] API tests documented in Postman
- [ ] Cross-portfolio isolation verified
- [ ] Code reviewed and merged
- [ ] Test cases: TC-Portfolio-01, TC-Portfolio-02 passed

Linked Requirements: CRYPTO-F-004

Test Cases: TC-Portfolio-01, TC-Portfolio-02

Components:

- Frontend: `PortfolioList.tsx`, `PortfolioForm.tsx`, `PortfolioCard.tsx`
- Backend: `portfolio_service.rs`, `portfolio_handlers.rs`
- Database: `portfolios` table

Dependencies:

- Authentication system must be completed
- Database connection pooling configured

Story 5: Manual Cryptocurrency Holdings Addition

Story ID: CRYPTO-US-005

Story Name: Add Manual Cryptocurrency Holdings

Story Type: Story

Epic Link: CRYPTO-EPIC-002

Priority: High

Story Points: 5

Sprint: Sprint 1

Assignee: Full Stack Team

Reporter: Product Owner

Labels: portfolio, holdings, frontend, backend

Summary: As a portfolio owner, I want to manually add my cryptocurrency holdings so that I can track assets I don't want to connect via exchange API.

Description: Implement manual holdings entry allowing users to add cryptocurrency with symbol, quantity, purchase price, and date. System validates inputs, calculates cost basis, and associates holdings with selected portfolio.

User Story (Narrative):

```
AS A portfolio owner
I WANT TO manually add my cryptocurrency holdings
SO THAT I can track assets from cold wallets or exchanges I don't want to connect
```

Acceptance Criteria:

```
GIVEN I am viewing a portfolio
WHEN I click "Add Holding" and select cryptocurrency
THEN I can enter symbol (autocomplete from top 500), quantity, purchase price, and date

GIVEN I enter valid holding details
WHEN I submit the form
THEN holding is added to portfolio with cost basis calculated automatically
```

```
GIVEN I enter invalid quantity (negative or zero)
WHEN I try to submit
THEN I see validation error "Quantity must be greater than 0"

GIVEN I select purchase date in the future
WHEN I submit the form
THEN I see error "Purchase date cannot be in the future"

GIVEN I have added a holding
WHEN I view portfolio holdings list
THEN I see symbol, quantity, purchase price, current price, current value, and unrealized
P&L

GIVEN I want to update a holding
WHEN I click edit and modify quantity or purchase price
THEN cost basis is recalculated and changes are saved

GIVEN I want to remove a holding
WHEN I click delete and confirm
THEN holding is permanently removed from portfolio

GIVEN I add multiple transactions for same asset
WHEN I view that asset
THEN I see aggregated position with average cost basis (FIFO/LIFO options)
```

Technical Implementation Details:

Frontend (React + TypeScript):

- Add holding form with autocomplete for symbols
- Quantity input with decimal support (up to 8 decimals)
- Purchase price input with currency selection
- Date picker for purchase date (default: today)
- Form validation with real-time feedback
- Holdings list table with sortable columns
- Edit/delete actions for each holding

Backend (Python — FastAPI + Pydantic + Decimal):

- Endpoints as above implemented with async SQLAlchemy and Pydantic schemas
- Pydantic models validate Decimal precision (use Python's Decimal with quantize)
- Symbol autocomplete backed by a seeded `cryptos` table and an async text search
- Cost basis calculations implemented in Python (decimal-aware) with unit tests

Pydantic example:

```python
from pydantic import BaseModel, condecimal, Field
from decimal import Decimal

class HoldingCreate(BaseModel):
  symbol: str
  quantity: condecimal(gt=0, max_digits=20, decimal_places=8)
```

```
  purchase_price: condecimal(gt=0, max_digits=20, decimal_places=8)
  purchase_date: str
```

Database (PostgreSQL):

```
CREATE TABLE holdings (
  id UUID PRIMARY KEY,
  portfolio_id UUID REFERENCES portfolios(id) ON DELETE CASCADE,
  symbol VARCHAR(20) NOT NULL,
  quantity DECIMAL(20, 8) NOT NULL CHECK (quantity > 0),
  purchase_price DECIMAL(20, 8) NOT NULL CHECK (purchase_price > 0),
  purchase_date DATE NOT NULL,
  purchase_currency VARCHAR(3) DEFAULT 'USD',
  notes TEXT,
  source VARCHAR(50) DEFAULT 'manual',
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_holdings_portfolio_id ON holdings(portfolio_id);
CREATE INDEX idx_holdings_symbol ON holdings(symbol);

CREATE TABLE transactions (
  id UUID PRIMARY KEY,
  holding_id UUID REFERENCES holdings(id) ON DELETE CASCADE,
  transaction_type VARCHAR(20) NOT NULL, -- 'buy', 'sell'
  quantity DECIMAL(20, 8) NOT NULL,
  price DECIMAL(20, 8) NOT NULL,
  transaction_date TIMESTAMP NOT NULL,
  created_at TIMESTAMP DEFAULT NOW()
);
```

Testing Requirements:

- Unit tests: Input validation (quantity, price, date) (Cargo test)
- Unit tests: Cost basis calculation logic (Cargo test)
- Component tests: Add holding form (Jest + React Testing Library)
- Integration test: Full add/edit/delete holding flow (Cypress)
- API tests: Holdings endpoints (Postman)
- Edge case tests: Decimal precision, large numbers, negative values
- Test: Symbol validation against cryptocurrency list

Definition of Done:

- [ ] Frontend holding form and list components completed
- [ ] Backend holdings CRUD endpoints implemented
- [ ] Database schema with constraints created
- [ ] Symbol autocomplete working with top 500 cryptos
- [ ] Cost basis calculation implemented and tested
- [ ] Input validation (client and server) working
- [ ] Unit tests passing (95%+ coverage)

- [ ] Integration tests passing
- [ ] API tests documented
- [ ] Edge cases tested (decimals, large numbers)
- [ ] Code reviewed and merged
- [ ] Test cases: TC-Portfolio-03, TC-Portfolio-04 passed

Linked Requirements: CRYPTO-F-005, CRYPTO-SR-005

Test Cases: TC-Portfolio-03, TC-Portfolio-04

Components:

- Frontend: `AddHoldingForm.tsx`, `HoldingsList.tsx`, `HoldingRow.tsx`
- Backend: `holdings_service.rs`, `cost_basis_calculator.rs`
- Database: `holdings` table, `transactions` table

Dependencies:

- Cryptocurrency symbol list (top 500) seeded in database
- Portfolio management system completed
- Decimal precision library configured

Story 6: Real-Time Cryptocurrency Price Tracking

Story ID: CRYPTO-US-006

Story Name: WebSocket-Based Real-Time Price Updates

Story Type: Story

Epic Link: CRYPTO-EPIC-002

Priority: High

Story Points: 8

Sprint: Sprint 1

Assignee: Backend + Frontend Team

Reporter: Technical Lead

Labels: market-data, websocket, real-time, backend, frontend, performance

---

Summary: As a portfolio manager, I want to see real-time prices for all my cryptocurrencies so that I can monitor my portfolio value throughout the day.

Description: Implement real-time price tracking using WebSocket connections to market data providers. Prices update within 10 seconds of market changes. System handles multiple currencies, API outages, and concurrent connections efficiently.

User Story (Narrative):

```
AS A portfolio manager
I WANT TO see real-time cryptocurrency prices updating automatically
```

```
SO THAT I can monitor my portfolio value without manual refreshing
```

Acceptance Criteria:

```
GIVEN I am viewing my portfolio dashboard
WHEN cryptocurrency prices change in the market
THEN I see updated prices within 10 seconds without page refresh

GIVEN I have holdings in multiple cryptocurrencies
WHEN I view my portfolio
THEN all prices update simultaneously via WebSocket

GIVEN the market data API experiences temporary outage
WHEN prices cannot be fetched
THEN system displays last known prices with timestamp and "stale" indicator

GIVEN I switch between portfolios
WHEN I navigate to different portfolio
THEN WebSocket subscribes to new symbols and unsubscribes from old ones

GIVEN I select different display currency (USD, EUR, GBP, INR)
WHEN I change currency preference
THEN all prices convert and display in selected currency

GIVEN WebSocket connection drops
WHEN network is restored
THEN connection automatically reconnects and resumes price updates

GIVEN 1000+ users are connected simultaneously
WHEN all receive price updates
THEN system handles load without >20% performance degradation
```

Technical Implementation Details:

Frontend (React + TypeScript):

- WebSocket client connection management
- Auto-reconnection logic with exponential backoff
- Price update reducer for state management
- Price display component with currency formatting
- "Last updated" timestamp display
- Loading/stale/error states for prices
- WebSocket hook for reusable connection logic

Backend (Python — FastAPI WebSockets + aiohttp + asyncio):

- FastAPI WebSocket endpoint `/ws/prices` supporting auth via query header or cookie
- Use `aiohttp` or `httpx` (async) to poll CoinGecko/market APIs and publish updates
- Price aggregation and short TTL caching in Redis
- Per-connection subscription management stored in Redis (or in-memory + Redis for scale)
- Broadcast updates to connected WebSocket clients with efficient asyncio tasks and fanout
- Fallback to polling when provider WebSocket is unavailable

Example WebSocket handler (simplified):

```
from fastapi import WebSocket

async def ws_prices(websocket: WebSocket):
  await websocket.accept()
  try:
    while True:
      data = await websocket.receive_json()
      # handle subscribe/unsubscribe
  except WebSocketDisconnect:
    pass
```

Database (Redis Cache):

```
Key: price:{symbol}:{currency}
Value: {
  "price": 47000.00,
  "change_24h": 2.5,
  "percent_change_24h": 5.32,
  "volume_24h": 28500000000,
  "last_updated": "2025-10-10T12:00:00Z"
}
TTL: 60 seconds

Key: ws_connection:{user_id}
Value: {subscribed_symbols: ["BTC", "ETH", "SOL"]}
TTL: Session duration
```

API Integration:

- Primary: CoinGecko API (free tier: 50 calls/min)
- Fallback: CoinMarketCap API
- Polling interval: 10 seconds for subscribed symbols
- Batch requests for multiple symbols

Testing Requirements:

- Unit tests: WebSocket message handling (Cargo test)
- Unit tests: Price formatting and currency conversion (Jest)
- Component tests: Price display updates (Jest + React Testing Library)
- Integration test: WebSocket connection and updates (Cypress)
- Load test: 1000+ concurrent WebSocket connections (JMeter)
- Performance test: Price update latency measurement
- Test: Reconnection logic on network failure
- Test: Graceful handling of API rate limits
- API tests: Market data endpoints (Postman)

Definition of Done:

- [ ] WebSocket server implemented in Rust
- [ ] WebSocket client implemented in React

- [ ] Market data API integration completed
- [ ] Redis caching layer configured
- [ ] Auto-reconnection logic working
- [ ] Currency conversion implemented
- [ ] Unit tests passing (95%+ coverage)
- [ ] Integration tests passing
- [ ] Load testing with 1000+ connections passed
- [ ] Latency requirements met (<10s updates)
- [ ] Fallback mechanisms tested
- [ ] Code reviewed and merged
- [ ] Test cases: TC-Market-01, TC-Market-02, TC-Perf-01 passed

Linked Requirements: CRYPTO-F-007, CRYPTO-NF-001, CRYPTO-NF-004

Test Cases: TC-Market-01, TC-Market-02, TC-Perf-01, TC-Scale-01

Components:

- Frontend: `WebSocketProvider.tsx`, `PriceDisplay.tsx`, `useWebSocket.ts`
- Backend: `websocket_server.rs`, `market_data_service.rs`, `price_aggregator.rs`
- Infrastructure: Redis cache

Dependencies:

- CoinGecko/CoinMarketCap API keys obtained
- Redis instance configured and accessible
- WebSocket infrastructure deployed
- Currency conversion rates API configured

Story 7: Portfolio Dashboard Overview

Story ID: CRYPTO-US-007

Story Name: Portfolio Dashboard with Real-Time Valuation

Story Type: Story

Epic Link: CRYPTO-EPIC-002

Priority: High

Story Points: 8

Sprint: Sprint 1

Assignee: Full Stack Team

Reporter: Product Owner

Labels: dashboard, frontend, backend, performance, analytics

Summary: As a portfolio owner, I want to see a dashboard with my total portfolio value and asset allocation so that I can quickly assess my investment status.

Description: Build comprehensive dashboard displaying total portfolio value, 24h gain/loss, asset allocation breakdown, and top holdings. Dashboard loads within 3 seconds and updates in real-time as prices change.

User Story (Narrative):

```
AS A portfolio owner
I WANT TO see a comprehensive dashboard of my portfolio
SO THAT I can quickly understand my current investment position
```

Acceptance Criteria:

```
GIVEN I log in successfully
WHEN dashboard loads
THEN I see total portfolio value, 24h change, and asset allocation within 3 seconds

GIVEN I have multiple portfolios
WHEN I select a specific portfolio from dropdown
THEN dashboard updates to show that portfolio's data

GIVEN cryptocurrency prices update in real-time
WHEN prices change
THEN dashboard recalculates and displays new values without manual refresh

GIVEN I have 10+ different assets
WHEN I view dashboard
THEN I see top 5 assets by value with allocation percentages

GIVEN I view asset allocation
WHEN I see the pie/donut chart
THEN percentages sum to 100% and colors are distinct

GIVEN I hover over any asset in the chart
WHEN I see tooltip
THEN it shows asset name, quantity, value, and percentage

GIVEN I am on mobile device
WHEN I view dashboard
THEN layout is responsive with critical info prioritized

GIVEN dashboard data is loading
WHEN page is rendering
THEN I see skeleton screens, not blank page
```

Technical Implementation Details:

Frontend (React + TypeScript):

- Dashboard container component with layout
- Portfolio value card with animated counters
- 24h change indicator with red/green color coding
- Asset allocation chart (Chart.js donut chart)
- Top holdings table with sorting capability

- Loading skeleton components
- Responsive grid layout (Material-UI Grid)
- Real-time value recalculation on price updates
- Currency selector dropdown

Backend (Python — FastAPI + async queries + Redis caching):

- `/api/dashboard` and `/api/portfolios/{id}/summary` implemented with async SQLAlchemy queries
- Use Redis to cache computed dashboard JSON (TTL ~30s) and to store precomputed price snapshots
- Aggregation and heavy calculations performed in async tasks where possible and cached
- Use connection pooling and optimized SQL with indexes as described in DB schema

Caching pattern: try Redis cache key `dashboard:{user_id}:{portfolio_id}` -> if miss, run async query, store JSON in Redis with TTL 30s.

Database (PostgreSQL + Redis):

PostgreSQL queries:

```
-- Portfolio summary with holdings
SELECT
  h.symbol,
  h.quantity,
  h.purchase_price,
  h.purchase_date,
  p.current_price,
  (h.quantity * p.current_price) as current_value,
  ((p.current_price - h.purchase_price) / h.purchase_price * 100) as gain_loss_percent
FROM holdings h
LEFT JOIN prices p ON h.symbol = p.symbol
WHERE h.portfolio_id = $1
ORDER BY current_value DESC;

-- Portfolio total value
SELECT
  SUM(h.quantity * p.current_price) as total_value
FROM holdings h
LEFT JOIN prices p ON h.symbol = p.symbol
WHERE h.portfolio_id = $1;
```

Redis cache:

```
Key: dashboard:{user_id}:{portfolio_id}
Value: {
  "total_value": 52000.00,
  "change_24h": 2000.00,
  "percent_change_24h": 4.0,
  "top_holdings": [...],
  "allocation": {...}
}
TTL: 30 seconds
```

Performance Optimization:

- Database connection pooling (20 connections)
- Query result caching in Redis
- Lazy loading for non-critical components
- Pagination for large holdings lists
- Code splitting for dashboard bundle

Testing Requirements:

- Unit tests: Value calculation logic (Cargo test)
- Unit tests: Percentage calculation accuracy (Cargo test)
- Component tests: Dashboard components (Jest + React Testing Library)
- Integration test: Full dashboard load and interaction (Cypress)
- Performance test: Dashboard load time with 100 assets (Lighthouse)
- Performance test: 95th percentile load time ≤3s (JMeter)
- API tests: Dashboard endpoints (Postman)
- Responsive design test: Mobile, tablet, desktop viewports
- Accessibility test: WCAG 2.1 AA compliance (axe-core)

Definition of Done:

- [ ] Frontend dashboard components completed
- [ ] Backend aggregation endpoints implemented
- [ ] Value calculation logic tested and accurate
- [ ] Real-time updates integrated with WebSocket
- [ ] Chart.js integration working smoothly
- [ ] Loading states and skeletons implemented
- [ ] Responsive design verified on all viewports
- [ ] Performance requirements met (<3s load)
- [ ] Unit tests passing (95%+ coverage)
- [ ] Integration tests passing
- [ ] Accessibility audit passed
- [ ] Code reviewed and merged
- [ ] Test cases: TC-Analytics-01, TC-Perf-01 passed

Linked Requirements: CRYPTO-F-010, CRYPTO-NF-001, CRYPTO-NF-003

Test Cases: TC-Analytics-01, TC-Perf-01, TC-Mobile-01

Components:

- Frontend: `Dashboard.tsx`, `PortfolioSummary.tsx`, `AssetAllocation.tsx`, `TopHoldings.tsx`
- Backend: `dashboard_service.rs`, `portfolio_aggregator.rs`
- Database: Optimized queries, Redis cache

Dependencies:

- Real-time price tracking (US-006) completed
- Portfolio and holdings management (US-004, US-005) completed
- Chart.js library configured
- Performance monitoring tools setup

Story 8: Exchange Account Connection Setup

Story ID: CRYPTO-US-008

Story Name: Secure Exchange API Connection (Binance & Coinbase)

Story Type: Story

Epic Link: CRYPTO-EPIC-002

Priority: High

Story Points: 13

Sprint: Sprint 1

Assignee: Backend + Security Team

Reporter: Technical Lead

Labels: exchange-integration, security, oauth, backend

Summary: As a user with existing exchange accounts, I want to securely connect my Binance/Coinbase account so that I can auto-import my transactions in future sprints.

Description: Implement OAuth 2.0 flow for exchange connections with AES-256 encrypted API key storage. This sprint focuses on connection setup and credential management; actual transaction import will be Sprint 2.

User Story (Narrative):

```
AS A user with exchange accounts
I WANT TO securely connect my Binance or Coinbase account
SO THAT the system can access my transaction history (future sprint)
```

Acceptance Criteria:

```
GIVEN I navigate to exchange connections page
WHEN I click "Connect Binance" or "Connect Coinbase"
THEN I am redirected to exchange OAuth authorization page

GIVEN I approve authorization on exchange
WHEN I am redirected back to the app
THEN my exchange connection is established and credentials are stored encrypted

GIVEN exchange API credentials are stored
WHEN system stores them in database
THEN credentials are encrypted using AES-256, never in plaintext

GIVEN I view connected exchanges
WHEN I see the list
THEN I see exchange name, connection status, last sync time, and "Disconnect" button

GIVEN I want to disconnect an exchange
WHEN I click disconnect and confirm
THEN API credentials are deleted from database permanently
```

```
GIVEN invalid API credentials are provided
WHEN system validates them
THEN error message shown and credentials not stored

GIVEN I connect an exchange
WHEN credentials are stored
THEN audit log records connection event with timestamp and IP

GIVEN I have read-only API scope
WHEN system validates permissions
THEN trading permissions are rejected, only read access allowed
```

Technical Implementation Details:

Frontend (React + TypeScript):

- Exchange connections page with provider cards
- OAuth flow initiation buttons
- Loading states during OAuth redirect
- Connection status indicators
- Disconnect confirmation dialog
- Error handling for failed connections
- List of connected exchanges with details

Backend (Python — FastAPI + authlib + cryptography):

- OAuth flows implemented using `authlib` (async client helpers)
- `/api/exchanges/connect/{provider}` and `/api/exchanges/callback/{provider}` endpoints
- Exchange connection records stored encrypted using `cryptography` (AES-GCM) with a master key (rotate annually)
- Credential validation called before persisting
- Read-only scope enforcement on the server side
- Audit logging written to `exchange_connection_audit` table

Encryption snippet (cryptography AES-GCM):

```python
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import os

key = os.getenv("MASTER_ENCRYPTION_KEY").encode()  # 32 bytes
aesgcm = AESGCM(key)
nonce = os.urandom(12)
ct = aesgcm.encrypt(nonce, b"plain-api-secret", None)
# store ct and nonce in DB
```

Database (PostgreSQL):

```sql
CREATE TABLE exchange_connections (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  exchange_name VARCHAR(50) NOT NULL, -- 'binance', 'coinbase'
```

```
  api_key_encrypted BYTEA NOT NULL,
  api_secret_encrypted BYTEA NOT NULL,
  encryption_nonce BYTEA NOT NULL,
  oauth_access_token_encrypted BYTEA,
  oauth_refresh_token_encrypted BYTEA,
  scopes TEXT[], -- ['read:accounts', 'read:transactions']
  connection_status VARCHAR(20) DEFAULT 'active', -- 'active', 'invalid', 'expired'
  last_sync_at TIMESTAMP,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_exchange_connections_user_id ON exchange_connections(user_id);

CREATE TABLE exchange_connection_audit (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  exchange_connection_id UUID REFERENCES exchange_connections(id),
  action VARCHAR(50) NOT NULL, -- 'connected', 'disconnected', 'validated'
  ip_address VARCHAR(45),
  user_agent TEXT,
  created_at TIMESTAMP DEFAULT NOW()
);
```

Encryption Implementation:

```
// AES-256-GCM encryption
use aes_gcm::{Aes256Gcm, Key, Nonce};
use aes_gcm::aead::{Aead, KeyInit};

// Master encryption key from environment (rotate annually)
const MASTER_KEY: &str = env!("ENCRYPTION_MASTER_KEY");

fn encrypt_api_key(plaintext: &str) -> (Vec<u8>, Vec<u8>) {
    let key = Key::<Aes256Gcm>::from_slice(MASTER_KEY.as_bytes());
    let cipher = Aes256Gcm::new(key);
    let nonce = Aes256Gcm::generate_nonce(&mut OsRng);
    let ciphertext = cipher.encrypt(&nonce, plaintext.as_bytes()).unwrap();
    (ciphertext, nonce.to_vec())
}
```

Exchange API Integration:

Binance OAuth:

- Authorization URL: `https://accounts.binance.com/oauth/authorize`
- Token URL: `https://accounts.binance.com/oauth/token`
- Scopes: `user:read`, `trade:read` (read-only)

Coinbase OAuth:

- Authorization URL: `https://www.coinbase.com/oauth/authorize`

- Token URL: `https://api.coinbase.com/oauth/token`
- Scopes: `wallet:accounts:read`, `wallet:transactions:read`

Security Measures:

- API keys never logged in plaintext
- Encryption at rest (AES-256-GCM)
- TLS 1.3 in transit
- CSRF protection for OAuth callbacks
- State parameter validation
- Scope restriction (read-only enforced)
- Rate limiting on connection attempts

Testing Requirements:

- Unit tests: Encryption/decryption functions (Cargo test)
- Unit tests: OAuth flow logic (Cargo test)
- Integration test: Full OAuth connection flow (Cypress)
- Integration test: Binance sandbox connection
- Integration test: Coinbase sandbox connection
- Security test: Verify no plaintext API keys in logs (grep)
- Security test: Encryption validation (OWASP ZAP)
- Security test: OAuth CSRF protection (manual testing)
- Penetration test: API key extraction attempts
- API tests: Exchange connection endpoints (Postman)

Definition of Done:

- [ ] OAuth 2.0 flow implemented for Binance and Coinbase
- [ ] AES-256 encryption service implemented and tested
- [ ] Frontend exchange connection UI completed
- [ ] Backend connection endpoints working
- [ ] Plaintext API key prevention verified (no logs)
- [ ] Audit logging capturing all connection events
- [ ] Read-only scope enforcement validated
- [ ] Database schema with encrypted columns created
- [ ] Unit tests passing (95%+ coverage)
- [ ] Integration tests with sandbox accounts passing
- [ ] Security audit completed and approved
- [ ] Penetration testing passed
- [ ] Documentation updated with security procedures
- [ ] Test cases: TC-Exchange-01, TC-Sec-02 passed

Linked Requirements: CRYPTO-F-006, CRYPTO-SR-001, CRYPTO-SR-004

Test Cases: TC-Exchange-01, TC-Sec-02, TC-Sec-05

Components:

- Frontend: `ExchangeConnections.tsx`, `OAuthCallback.tsx`
- Backend: `oauth_service.rs`, `encryption_service.rs`, `binance_client.rs`, `coinbase_client.rs`

- Database: `exchange_connections` table, `exchange_connection_audit` table

Dependencies:

- Binance OAuth app credentials (Client ID/Secret)
- Coinbase OAuth app credentials
- Encryption master key generated and secured
- Audit logging infrastructure ready

Story 9: User Security Settings Management

Story ID: CRYPTO-US-009

Story Name: Security Settings and Device Session Management

Story Type: Story

Epic Link: CRYPTO-EPIC-001

Priority: Medium

Story Points: 8

Sprint: Sprint 1

Assignee: Full Stack + Security Team

Reporter: Security Lead

Labels: security, user-management, frontend, backend

Summary: As a user, I want to manage my security preferences so that I can configure authentication and privacy settings according to my needs.

Description: Build comprehensive security settings page allowing users to change passwords, manage 2FA, view connected devices/sessions, and review security event logs with email notifications for sensitive changes.

User Story (Narrative):

```
AS A security-conscious user
I WANT TO manage all my security settings in one place
SO THAT I can maintain control over my account security
```

Acceptance Criteria:

```
GIVEN I navigate to security settings
WHEN page loads
THEN I see sections for password, 2FA, sessions, and security log

GIVEN I want to change password
WHEN I enter current password, new password, and confirmation
THEN password is updated and I receive email notification

GIVEN I enter incorrect current password
WHEN I try to change password
```

```
THEN I see error "Current password is incorrect"

GIVEN I want to disable 2FA
WHEN I click disable and confirm with password + 2FA code
THEN 2FA is disabled and I receive warning email

GIVEN I view connected sessions
WHEN I see the list
THEN each shows device type, browser, location, IP, last activity time

GIVEN I see suspicious session
WHEN I click "Log out" on that session
THEN that session is immediately terminated

GIVEN I click "Log out all other devices"
WHEN I confirm
THEN all sessions except current are terminated

GIVEN I view security event log
WHEN I see the list
THEN it shows login attempts, password changes, 2FA toggles with timestamps

GIVEN I make any security change
WHEN change is successful
THEN I receive email notification about the change
```

Technical Implementation Details:

Frontend (React + TypeScript):

- Security settings page with tabbed layout
- Password change form with validation
- 2FA management section (enable/disable/regenerate backup codes)
- Active sessions table with device info
- Security event log timeline
- Confirmation dialogs for sensitive actions
- Email notification preferences

Backend (Python — FastAPI + user-agents + geoip2):

- Endpoints as above implemented in FastAPI (async)
- Device fingerprinting using `user-agents` package and parsing User-Agent strings
- IP geolocation via MaxMind `geoip2` or a geo-IP HTTP API
- Security events logged in `security_events` table asynchronously
- Email notifications sent via SendGrid or SMTP (async)

Use `python-user-agents` to extract device type and browser from user-agent string, and `geoip2` to resolve IP to location.

Database (PostgreSQL):

```
-- Sessions table already exists from US-003
```

```sql
CREATE TABLE security_events (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  event_type VARCHAR(50) NOT NULL, -- 'login_success', 'login_failed', 'password_changed',
'2fa_enabled', '2fa_disabled', 'session_terminated'
  ip_address VARCHAR(45),
  user_agent TEXT,
  location VARCHAR(255), -- City, Country from IP geolocation
  metadata JSONB,
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_security_events_user_id ON security_events(user_id);
CREATE INDEX idx_security_events_created_at ON security_events(created_at DESC);

ALTER TABLE users ADD COLUMN password_changed_at TIMESTAMP;
ALTER TABLE users ADD COLUMN last_security_review_at TIMESTAMP;
```

Email Templates:

- Password changed notification
- 2FA enabled/disabled notification
- Suspicious login attempt alert
- Session terminated notification

Testing Requirements:

- Unit tests: Password validation and hashing (Cargo test)
- Component tests: Security settings forms (Jest + React Testing Library)
- Integration test: Password change flow (Cypress)
- Integration test: Session management (Cypress)
- Integration test: Email notifications (mock SMTP)
- API tests: Security endpoints (Postman)
- Test: Concurrent session termination
- Test: Security log pagination and filtering

Definition of Done:

- [ ] Frontend security settings UI completed
- [ ] Backend security management endpoints implemented
- [ ] Password change with email notification working
- [ ] Session management fully functional
- [ ] Security event logging comprehensive
- [ ] Device fingerprinting implemented
- [ ] IP geolocation integration working
- [ ] Email notification templates created
- [ ] Unit tests passing (95%+ coverage)
- [ ] Integration tests passing
- [ ] Email delivery tested
- [ ] Code reviewed and merged

- [ ] Test cases: TC-Sec-04, TC-Sec-05 passed

Linked Requirements: CRYPTO-SR-003, CRYPTO-SR-004

Test Cases: TC-Sec-04, TC-Sec-05

Components:

- Frontend: `SecuritySettings.tsx`, `SessionManagement.tsx`, `SecurityLog.tsx`
- Backend: `security_service.rs`, `session_manager.rs`, `event_logger.rs`
- Database: `security_events` table

Dependencies:

- Email service configured (SendGrid)
- IP geolocation API configured
- User-Agent parser library installed

Story 10: Automated Backup Infrastructure

Story ID: CRYPTO-US-010

Story Name: Daily Encrypted Database Backup System

Story Type: Story

Epic Link: CRYPTO-EPIC-002

Priority: High

Story Points: 8

Sprint: Sprint 1

Assignee: DevOps + Backend Team

Reporter: Technical Lead

Labels: infrastructure, backup, security, devops

Summary: As the system, we need automated encrypted backup mechanisms so that user data is protected and recoverable in case of incidents.

Description: Establish automated daily backup infrastructure with AES-256 encryption, 90-day retention, point-in-time recovery capability, and monitoring with alerts for failed backups.

User Story (Narrative):

```
AS THE system
WE NEED automated encrypted backups
SO THAT user data can be recovered in case of disasters or data corruption
```

Acceptance Criteria:

```
GIVEN it is 2:00 AM UTC daily
WHEN backup job runs
```

```
THEN full database backup is created and encrypted with AES-256

GIVEN backup is created
WHEN it is stored
THEN it is uploaded to AWS S3 with versioning enabled

GIVEN backup is uploaded
WHEN integrity check runs
THEN checksum is verified and stored in metadata

GIVEN backup fails for any reason
WHEN failure is detected
THEN admin receives immediate alert via email and Slack

GIVEN backups are retained
WHEN they are older than 90 days
THEN they are automatically deleted per retention policy

GIVEN point-in-time recovery is needed
WHEN restore request is made
THEN any backup from last 90 days can be restored

GIVEN backup/restore process is tested
WHEN monthly DR drill runs
THEN recovery completes successfully within 4 hours
```

Technical Implementation Details:

Backup system (Python — Celery + boto3 + cryptography):

Daily backups are implemented as Celery tasks. Use `pg_dump` or `pg_dumpall` called from within a controlled environment, encrypt the dump with `cryptography` (AES-GCM) and upload to S3 via `boto3`.

Example Celery task (simplified):

```python
from celery import Celery
import subprocess, os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import boto3

celery = Celery("backup", broker=os.getenv("CELERY_BROKER_URL"))

@celery.task
def perform_backup():
  date = subprocess.check_output(["date", "+%Y%m%d_%H%M%S"]).decode().strip()
  dump_file = f"crypto_portfolio_{date}.sql"
  subprocess.check_call(["pg_dump", os.getenv("DB_NAME"), "-f", dump_file])
  key = os.getenv("MASTER_ENCRYPTION_KEY").encode()
  aesgcm = AESGCM(key)
  nonce = os.urandom(12)
  with open(dump_file, "rb") as f:
    ct = aesgcm.encrypt(nonce, f.read(), None)
```

```
s3 = boto3.client("s3")
s3.put_object(Bucket=os.getenv("S3_BUCKET"), Key=dump_file + ".enc", Body=nonce + ct)
os.remove(dump_file)
```

Monitoring and alerts are wired into Celery failure handlers and CloudWatch/SNS as described.

AWS S3 Configuration:

- Bucket: `crypto-portfolio-backups`
- Versioning: Enabled
- Encryption: Server-side (AES-256)
- Lifecycle policy: Delete after 90 days
- Access: Private, admin IAM role only
- Region: Multi-region replication for redundancy

Monitoring & Alerting:

```
# CloudWatch Alarm
Metric: BackupFailure
Threshold: >= 1
Period: 1 hour
Actions:
  - SNS Topic: admin-alerts
  - Email: devops-team@company.com
  - Slack webhook: #infrastructure-alerts
```

Recovery Procedures:

1. Identify backup file from S3
2. Download encrypted backup
3. Verify checksum
4. Decrypt using encryption key
5. Create new database instance (if needed)
6. Restore using `psql` or `pg_restore`
7. Validate data integrity
8. Switch application to restored database

Testing Requirements:

- Unit tests: Encryption/decryption functions (Cargo test)
- Integration test: Full backup creation and upload
- Integration test: Checksum verification
- Integration test: Full restore process
- Test: Backup job scheduling (cron)
- Test: Alert notification on failure
- Monthly DR drill: Full recovery exercise
- Test: 90-day retention policy enforcement

Definition of Done:

- ☐ Backup script/service implemented
- ☐ AWS S3 bucket configured with policies
- ☐ Encryption working (AES-256 verified)

- [ ] Daily cron job scheduled and running
- [ ] Checksum validation implemented
- [ ] Monitoring and alerting configured
- [ ] Admin notification system working
- [ ] Recovery documentation complete
- [ ] Full backup-restore tested successfully
- [ ] Monthly DR drill scheduled
- [ ] Retention policy automated
- [ ] Integration tests passing
- [ ] Security review completed
- [ ] Test cases: TC-Backup-01 passed

Linked Requirements: CRYPTO-F-020, CRYPTO-NF-002

Test Cases: TC-Backup-01, TC-Monitoring-01

Components:

- Scripts: `backup.sh`, `restore.sh`
- Backend: `backup_service.rs`, `recovery_service.rs`
- Infrastructure: AWS S3, CloudWatch, SNS

Dependencies:

- AWS account with S3 access
- PostgreSQL backup utilities installed
- OpenSSL for encryption
- CloudWatch monitoring configured
- Admin email/Slack configured

SPRINT 1 SUMMARY

Total Story Points: 76

Sprint Duration: 2 weeks (Oct 1-15, 2025)

Team Capacity Planning:

- Frontend Team (2 devs): 40 points
- Backend Team (2 devs): 45 points
- Overlap/Collaboration: Multiple stories require both

Key Milestones:

- **Day 3:** Authentication infrastructure (US-001, US-002, US-003)
- **Day 7:** Portfolio management (US-004, US-005)
- **Day 10:** Real-time data & dashboard (US-006, US-007)
- **Day 12:** Exchange connections & security (US-008, US-009)
- **Day 14:** Backup infrastructure & testing (US-010)

Tech Stack Summary:

- **Frontend:** React 18 + TypeScript + Material-UI + Chart.js (unchanged)
- **Backend:** Python 3.11+ — FastAPI (async), Pydantic, Async SQLAlchemy 2.0, Celery for background tasks
- **Database:** PostgreSQL 15 + Redis 7
- **Testing:** pytest (async support), httpx (for async integration tests), Cypress (frontend), locust (load tests), Postman, OWASP ZAP
- **Infrastructure:** Docker, AWS S3, CloudWatch

Recommended Python backend directory structure

```
src/backend/
├─ app/
│  ├─ main.py            # FastAPI app / startup
│  ├─ api/
│  │  ├─ auth.py
│  │  ├─ portfolios.py
│  │  └─ websocket.py
│  ├─ models/            # SQLAlchemy models (async)
│  ├─ schemas/           # Pydantic request/response schemas
│  ├─ services/          # business logic, encryption, oauth clients
│  └─ tasks.py           # Celery task registrations
├─ alembic/              # migrations
├─ tests/                # pytest tests
├─ requirements.txt
└─ pyproject.toml
```

Testing Strategy:

- Unit tests throughout development (Jest + Cargo test)
- Component tests for React (React Testing Library)
- Integration tests end-of-sprint (Cypress)
- API tests continuous (Postman collections)
- Security testing parallel (OWASP ZAP)
- Performance testing Day 12-13 (JMeter)
- Load testing Day 13 (1000+ concurrent users)

Definition of Done (Sprint Level):

- [ ] All 10 user stories completed and accepted
- [ ] 95%+ code coverage achieved
- [ ] All high-priority test cases passed
- [ ] Security audit completed with no critical issues
- [ ] Performance requirements met (<3s dashboard load)
- [ ] API documentation updated (Postman collections)
- [ ] Code reviewed and merged to develop branch
- [ ] Sprint demo prepared for stakeholders

Risks:

1. **Exchange OAuth complexity** - Mitigation: Start early (Day 8), sandbox testing
2. **WebSocket scalability** - Mitigation: Load testing Day 11, Redis optimization
3. **Security testing delays** - Mitigation: Parallel security testing throughout
4. **Database performance** - Mitigation: Query optimization, connection pooling

Next Sprint Preview (Sprint 2):

- Exchange transaction import automation
- Historical price charts with multiple timeframes
- Performance analytics (P&L calculations)
- Price alerts and notifications
- Advanced portfolio analytics
- Tax calculation foundation