Sprint 1: Core Foundation - JIRA Epic & Stories

Python Backend Migration

Sprint 1 - Quick Reference

EPIC 1: Authentication & Core Infrastructure (34 pts)

- **US-001**: User Registration with Email Verification (5 pts)
- **US-002**: TOTP-Based 2FA Implementation (8 pts)
- **US-003**: JWT-Based Session Management (8 pts)
- **US-009**: Security Settings & Device Management (8 pts)
- **US-010**: Daily Encrypted Backup System (8 pts)

EPIC 2: Portfolio Management Core (42 pts)

- **US-004**: Create Multiple Portfolios (5 pts)
- **US-005**: Add Manual Holdings (5 pts)
- **US-006**: Real-Time Price Tracking (WebSocket) (8 pts)
- **US-007**: Portfolio Dashboard Overview (8 pts)
- **US-008**: Exchange Connection Setup (Binance/Coinbase) (13 pts)

Updated Tech Stack

Backend Stack Migration

Layer	Tool	Reason
Framework	**FastAPI**	Async, production-grade, OpenAPI standard, WebSockets & JWT support
ORM	**SQLAlchemy 2.0 + Alembic**	Mature, async support, handles complex queries
Database	**PostgreSQL 15**	Same as before, fully compatible
Cache	**Redis 7**	Reuse current session + caching system
Auth	**PyJWT + passlib + pyotp**	Replaces Axum JWT and TOTP flow
Background tasks	**Celery + Redis broker**	For async email verification, backup jobs
Email	**SendGrid API via sendgrid-python**	Retain same email delivery mechanism
WebSockets	**FastAPI WebSockets / Starlette**	Real-time updates for prices
Encryption	**cryptography (AES-GCM)**	For API key encryption (Binance/Coinbase)
Testing	**pytest + httpx + locust**	Python testing ecosystem

Deployment	**Docker + Uvicorn + Gunicorn**	Clean container setup

Frontend Stack (Unchanged)

- **React 18 + TypeScript**
- **Material-UI**
- **Chart.js**
- **Formik** for form validation
- **Axios** for API calls

Component Mapping: Rust → Python

Feature	Rust (Axum)	Python (FastAPI)
`/api/auth/register`	`auth_service.rs`	`routers/auth.py` using `fastapi-mail` and `sqlalchemy`
`/api/auth/2fa/*`	`totp_service.rs`	`routers/2fa.py` with `pyotp`
`/api/auth/login`, JWT	`jwt_service.rs`	`routers/auth.py` with `PyJWT`
WebSocket `/ws/prices`	`websocket_server.rs`	`routers/ws_prices.py` using `FastAPI WebSocket`
Portfolio CRUD	`portfolio_service.rs`	`routers/portfolio.py`
Holdings CRUD	`holdings_service.rs`	`routers/holdings.py`
Exchange OAuth	`oauth_service.rs`	`routers/exchange.py` with `requests-oauthlib`
Encryption	`encryption_service.rs`	`services/encryption.py` using `cryptography.fernet`
Backup job	`backup_service.rs`	`tasks/backup.py` using Celery periodic task
Security log	`event_logger.rs`	`services/security_log.py` with SQLAlchemy models

Backend Directory Structure

```
backend/
├── app/
│   ├── main.py
│   ├── core/
│   │   ├── config.py
│   │   ├── security.py
│   │   ├── jwt_handler.py
│   │   └── email_utils.py
│   ├── db/
│   │   ├── base.py
│   │   ├── models/
│   │   │   ├── user.py
│   │   │   ├── portfolio.py
│   │   │   ├── holdings.py
│   │   │   ├── exchange.py
```

```
│   │   │   └── security_events.py
│   │   └── session.py
│   ├── routers/
│   │   ├── auth.py
│   │   ├── twofa.py
│   │   ├── portfolio.py
│   │   ├── holdings.py
│   │   ├── dashboard.py
│   │   ├── exchange.py
│   │   ├── security.py
│   │   └── ws_prices.py
│   ├── services/
│   │   ├── encryption.py
│   │   ├── market_data.py
│   │   ├── backup.py
│   │   └── audit.py
│   └── tasks/
│       ├── backup.py
│       └── email.py
├── alembic/
├── tests/
│   ├── test_auth.py
│   ├── test_portfolio.py
│   └── test_ws.py
├── Dockerfile
├── pyproject.toml
└── requirements.txt
```

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.

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), **FastAPI + Python** (Backend), 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.

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, **FastAPI + Python**, PostgreSQL, Redis, WebSocket, Market Data APIs (CoinGecko/CoinMarketCap)

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 (FastAPI + Python)

- `POST /api/auth/register` endpoint
- Email validation logic using Pydantic
- Password hashing using `passlib` with bcrypt (cost factor 12)

- JWT token generation for email verification using `PyJWT`
- Database transaction for user creation with SQLAlchemy
- SendGrid integration via `sendgrid-python` for email delivery

Example Implementation:

```python
# routers/auth.py
from fastapi import APIRouter, HTTPException, BackgroundTasks
from pydantic import BaseModel, EmailStr
from passlib.context import CryptContext
from core.jwt_handler import create_verification_token
from core.email_utils import send_verification_email

router = APIRouter(prefix="/api/auth", tags=["authentication"])
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

class RegisterRequest(BaseModel):
    email: EmailStr
    password: str  # min 8 chars, validated

@router.post("/register")
async def register(data: RegisterRequest, background_tasks: BackgroundTasks):
    # Check if user exists
    # Hash password
    password_hash = pwd_context.hash(data.password)
    # Create user in DB
    # Generate verification token
    token = create_verification_token(user.id)
    # Send email asynchronously
    background_tasks.add_task(send_verification_email, data.email, token)
    return {"message": "Registration successful. Please verify your email."}
```

Database (PostgreSQL with SQLAlchemy)

```python
# db/models/user.py
from sqlalchemy import Column, String, Boolean, DateTime
from sqlalchemy.dialects.postgresql import UUID
import uuid
from datetime import datetime

class User(Base):
    __tablename__ = "users"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    email = Column(String(255), unique=True, nullable=False, index=True)
    password_hash = Column(String(255), nullable=False)
    email_verified = Column(Boolean, default=False)
    verification_token = Column(String(255))
    verification_expires_at = Column(DateTime)
    created_at = Column(DateTime, default=datetime.utcnow)
```

Testing Requirements

- Unit tests: Email validation, password strength check (pytest)
- Unit tests: Password hashing, token generation (pytest)
- Integration test: Full registration flow (pytest + httpx)
- API test: Registration endpoint validation (pytest or 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 (Alembic)
- [ ] 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: `routers/auth.py`, `core/email_utils.py`
- 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 (FastAPI + Python)

- TOTP library integration (`pyotp` library)
- `POST /api/auth/2fa/setup` endpoint
- `POST /api/auth/2fa/verify` endpoint

- `POST /api/auth/2fa/backup-codes` endpoint
- Secret key generation and secure storage (AES-256 encrypted)
- Rate limiting middleware using `slowapi` (max 5 attempts per 15 min)
- Backup code generation (10 codes using `secrets` module)

Example Implementation:

```
# routers/2fa.py
from fastapi import APIRouter, Depends, HTTPException
import pyotp
import qrcode
from io import BytesIO
import base64

router = APIRouter(prefix="/api/auth/2fa", tags=["2fa"])

@router.post("/setup")
async def setup_2fa(current_user: User = Depends(get_current_user)):
    # Generate TOTP secret
    secret = pyotp.random_base32()

    # Create TOTP URI
    totp_uri = pyotp.totp.TOTP(secret).provisioning_uri(
        name=current_user.email,
        issuer_name="CryptoPortfolio"
    )

    # Generate QR code
    qr = qrcode.make(totp_uri)
    buffered = BytesIO()
    qr.save(buffered, format="PNG")
    qr_code = base64.b64encode(buffered.getvalue()).decode()

    # Store encrypted secret (don't enable yet)
    # Return QR code and secret for manual entry
    return {
        "qr_code": f"{qr_code}",
        "secret": secret,
        "backup_codes": []  # Generated after verification
    }

@router.post("/verify")
async def verify_2fa(code: str, current_user: User = Depends(get_current_user)):
    # Verify TOTP code
    totp = pyotp.TOTP(current_user.totp_secret_decrypted)

    if not totp.verify(code, valid_window=1):
        raise HTTPException(status_code=400, detail="Invalid 2FA code")

    # Enable 2FA
    current_user.totp_enabled = True
```

```
    # Generate backup codes
    backup_codes = generate_backup_codes(10)

    return {"message": "2FA enabled", "backup_codes": backup_codes}
```

Database (PostgreSQL with SQLAlchemy)

```
# db/models/user.py - additions
class User(Base):
    # ... existing fields ...
    totp_secret = Column(String(255))  # Encrypted
    totp_enabled = Column(Boolean, default=False)

# db/models/backup_codes.py
class BackupCode(Base):
    __tablename__ = "backup_codes"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey("users.id"))
    code_hash = Column(String(255), nullable=False)
    used = Column(Boolean, default=False)
    created_at = Column(DateTime, default=datetime.utcnow)

# db/models/login_attempts.py
class LoginAttempt(Base):
    __tablename__ = "login_attempts"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey("users.id"))
    ip_address = Column(String(45))
    attempt_type = Column(String(50))
    success = Column(Boolean)
    attempted_at = Column(DateTime, default=datetime.utcnow)
```

Testing Requirements

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

Definition of Done

- [ ] TOTP library (`pyotp`) 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 (`slowapi`)
- [ ] Database schema updated via Alembic
- [ ] 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: `routers/2fa.py` , `core/security.py`
- Database: `totp_secret` column, `backup_codes` table, `login_attempts` table

Dependencies:

- `pyotp` library installed
- `qrcode` library for QR generation
- Rate limiting infrastructure ready (`slowapi`)

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 (FastAPI + Python)

- `POST /api/auth/login` endpoint
- JWT token generation with claims (user_id, exp, iat) using `PyJWT`
- JWT middleware for protected routes (FastAPI dependencies)
- Redis session storage for quick lookups
- `POST /api/auth/logout` endpoint
- `GET /api/auth/sessions` endpoint (list sessions)
- `DELETE /api/auth/sessions/{id}` endpoint (revoke session)
- Rate limiting middleware using `slowapi` (5 failed attempts = 15 min lock)
- Audit logging for all authentication events

Example Implementation:

```python
# core/jwt_handler.py
from datetime import datetime, timedelta
import jwt
from fastapi import HTTPException, Security
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials

SECRET_KEY = "your-secret-key"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
MAX_SESSION_HOURS = 24

security = HTTPBearer()

def create_access_token(user_id: str, session_id: str):
    payload = {
        "user_id": user_id,
        "session_id": session_id,
        "exp": datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES),
        "iat": datetime.utcnow(),
        "max_exp": datetime.utcnow() + timedelta(hours=MAX_SESSION_HOURS)
    }
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def verify_token(credentials: HTTPAuthorizationCredentials = Security(security)):
    try:
        payload = jwt.decode(credentials.credentials, SECRET_KEY, algorithms=[ALGORITHM])

        # Check max session time
        if datetime.fromtimestamp(payload["max_exp"]) < datetime.utcnow():
            raise HTTPException(status_code=401, detail="Session expired")

        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token expired")
    except jwt.JWTError:
        raise HTTPException(status_code=401, detail="Invalid token")

# routers/auth.py
from slowapi import Limiter
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)

@router.post("/login")
@limiter.limit("5/15minutes")
async def login(request: Request, credentials: LoginRequest):
    # Verify credentials
    # Verify 2FA if enabled
    # Create session in Redis
    # Generate JWT token
```

```
    # Log successful login
    return {"access_token": token, "token_type": "bearer"}
```

Database (PostgreSQL + Redis)

PostgreSQL:

```
# db/models/session.py
class Session(Base):
    __tablename__ = "sessions"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey("users.id"))
    token_hash = Column(String(255), nullable=False)
    device_info = Column(JSON)
    ip_address = Column(String(45))
    last_activity = Column(DateTime)
    expires_at = Column(DateTime)
    created_at = Column(DateTime, default=datetime.utcnow)

    __table_args__ = (
        Index('idx_sessions_user_id', 'user_id'),
        Index('idx_sessions_expires_at', 'expires_at'),
    )
```

Redis:

```
# Session cache structure
Key: f"session:{token_hash}"
Value: {
    "user_id": str,
    "expires_at": timestamp,
    "last_activity": timestamp
}
TTL: 24 hours
```

Testing Requirements

- Unit tests: JWT generation and validation (pytest)
- Unit tests: Session timeout logic (pytest)
- Component tests: Login form validation (Jest)
- Integration test: Full login and logout flow (pytest + httpx)
- Integration test: Session timeout behavior (pytest)
- API tests: Authentication endpoints (Postman)
- Load test: Concurrent login requests (locust)
- 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 (`slowapi`)
- [ ] 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: `core/jwt_handler.py` , `middleware/session.py` , `services/audit.py`
- Database: `sessions` table, Redis session cache

Dependencies:

- `PyJWT` library
- 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 (FastAPI + Python)

- `GET /api/portfolios` endpoint (list all user portfolios)
- `POST /api/portfolios` endpoint (create portfolio)
- `GET /api/portfolios/{id}` endpoint (single portfolio details)
- `PUT /api/portfolios/{id}` endpoint (update portfolio)
- `DELETE /api/portfolios/{id}` endpoint (delete with cascade)
- `POST /api/portfolios/{id}/set-default` endpoint

- Validation: Name required, length limits, XSS prevention using Pydantic

Example Implementation:

```python
# routers/portfolio.py
from fastapi import APIRouter, Depends, HTTPException
from pydantic import BaseModel, constr
from sqlalchemy.orm import Session

router = APIRouter(prefix="/api/portfolios", tags=["portfolios"])

class PortfolioCreate(BaseModel):
    name: constr(min_length=1, max_length=100)
    description: str | None = None

class PortfolioUpdate(BaseModel):
    name: constr(min_length=1, max_length=100) | None = None
    description: str | None = None

@router.post("/")
async def create_portfolio(
    portfolio: PortfolioCreate,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    new_portfolio = Portfolio(
        user_id=current_user.id,
        name=portfolio.name,
        description=portfolio.description,
        is_default=False
    )
    db.add(new_portfolio)
    db.commit()
    db.refresh(new_portfolio)
    return new_portfolio

@router.get("/")
async def list_portfolios(
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    portfolios = db.query(Portfolio).filter(
        Portfolio.user_id == current_user.id
    ).order_by(Portfolio.is_default.desc(), Portfolio.created_at.desc()).all()
    return portfolios
```

Database (PostgreSQL with SQLAlchemy)

```python
# db/models/portfolio.py
from sqlalchemy import Column, String, Boolean, DateTime, Text, UniqueConstraint
from sqlalchemy.dialects.postgresql import UUID
```

```python
import uuid
from datetime import datetime

class Portfolio(Base):
    __tablename__ = "portfolios"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey("users.id", ondelete="CASCADE"),
nullable=False)
    name = Column(String(100), nullable=False)
    description = Column(Text)
    is_default = Column(Boolean, default=False)
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    __table_args__ = (
        Index('idx_portfolios_user_id', 'user_id'),
        UniqueConstraint('user_id', 'is_default', name='uq_user_default_portfolio',
                         postgresql_where=Column('is_default') == True),
    )
```

Testing Requirements

- Unit tests: Portfolio validation logic (pytest)
- Component tests: Portfolio form validation (Jest)
- Component tests: Portfolio list rendering (Jest + React Testing Library)
- Integration test: Full portfolio CRUD flow (pytest + httpx)
- 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 (Alembic migration)
- [ ] Validation rules enforced (Pydantic models)
- [ ] Unit tests written and 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: `routers/portfolio.py` , `db/models/portfolio.py`
- Database: `portfolios` table

Dependencies:

- Authentication system must be completed
- Database connection pooling configured (SQLAlchemy)

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 (FastAPI + Python)

- `POST /api/portfolios/{id}/holdings` endpoint (add holding)
- `GET /api/portfolios/{id}/holdings` endpoint (list holdings)
- `PUT /api/portfolios/{id}/holdings/{holding_id}` endpoint (update)
- `DELETE /api/portfolios/{id}/holdings/{holding_id}` endpoint
- Symbol validation against supported crypto list
- Cost basis calculation logic (FIFO/LIFO methods)
- Input sanitization and validation using Pydantic

Example Implementation:

```
# routers/holdings.py
from fastapi import APIRouter, Depends, HTTPException
from pydantic import BaseModel, condecimal, constr
from decimal import Decimal
from datetime import date, datetime

router = APIRouter(prefix="/api/portfolios", tags=["holdings"])

class HoldingCreate(BaseModel):
    symbol: constr(min_length=1, max_length=20)
    quantity: condecimal(gt=0, max_digits=20, decimal_places=8)
    purchase_price: condecimal(gt=0, max_digits=20, decimal_places=8)
    purchase_date: date
    purchase_currency: str = "USD"
    notes: str | None = None
```

```
    @validator('purchase_date')
    def validate_purchase_date(cls, v):
        if v > date.today():
            raise ValueError('Purchase date cannot be in the future')
        return v

@router.post("/{portfolio_id}/holdings")
async def add_holding(
    portfolio_id: UUID,
    holding: HoldingCreate,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    # Verify portfolio ownership
    portfolio = db.query(Portfolio).filter(
        Portfolio.id == portfolio_id,
        Portfolio.user_id == current_user.id
    ).first()

    if not portfolio:
        raise HTTPException(status_code=404, detail="Portfolio not found")

    # Validate symbol exists in crypto list
    if not await validate_crypto_symbol(holding.symbol):
        raise HTTPException(status_code=400, detail="Invalid cryptocurrency symbol")

    # Create holding
    new_holding = Holding(
        portfolio_id=portfolio_id,
        symbol=holding.symbol.upper(),
        quantity=holding.quantity,
        purchase_price=holding.purchase_price,
        purchase_date=holding.purchase_date,
        purchase_currency=holding.purchase_currency,
        notes=holding.notes,
        source="manual"
    )

    db.add(new_holding)
    db.commit()
    db.refresh(new_holding)

    return new_holding
```

Database (PostgreSQL with SQLAlchemy)

```
# db/models/holding.py
from sqlalchemy import Column, String, DECIMAL, Date, Text, CheckConstraint
from sqlalchemy.dialects.postgresql import UUID
import uuid
```

```python
from datetime import datetime

class Holding(Base):
    __tablename__ = "holdings"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    portfolio_id = Column(UUID(as_uuid=True), ForeignKey("portfolios.id",
ondelete="CASCADE"), nullable=False)
    symbol = Column(String(20), nullable=False)
    quantity = Column(DECIMAL(20, 8), nullable=False)
    purchase_price = Column(DECIMAL(20, 8), nullable=False)
    purchase_date = Column(Date, nullable=False)
    purchase_currency = Column(String(3), default="USD")
    notes = Column(Text)
    source = Column(String(50), default="manual")
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    __table_args__ = (
        CheckConstraint('quantity > 0', name='check_quantity_positive'),
        CheckConstraint('purchase_price > 0', name='check_price_positive'),
        Index('idx_holdings_portfolio_id', 'portfolio_id'),
        Index('idx_holdings_symbol', 'symbol'),
    )

# db/models/transaction.py
class Transaction(Base):
    __tablename__ = "transactions"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    holding_id = Column(UUID(as_uuid=True), ForeignKey("holdings.id", ondelete="CASCADE"))
    transaction_type = Column(String(20), nullable=False)  # 'buy', 'sell'
    quantity = Column(DECIMAL(20, 8), nullable=False)
    price = Column(DECIMAL(20, 8), nullable=False)
    transaction_date = Column(DateTime, nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow)
```

Testing Requirements

- Unit tests: Input validation (quantity, price, date) (pytest)
- Unit tests: Cost basis calculation logic (pytest)
- Component tests: Add holding form (Jest + React Testing Library)
- Integration test: Full add/edit/delete holding flow (pytest + httpx)
- 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 (Alembic)

- [ ] Symbol autocomplete working with top 500 cryptos
- [ ] Cost basis calculation implemented and tested
- [ ] Input validation (Pydantic models) 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: `routers/holdings.py`, `services/cost_basis.py`
- Database: `holdings` table, `transactions` table

Dependencies:

- Cryptocurrency symbol list (top 500) seeded in database
- Portfolio management system completed
- Python `decimal` module for precision

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 (FastAPI + Python)

- WebSocket server endpoint `/ws/prices`
- Connection authentication via JWT token
- Market data service connecting to CoinGecko/CoinMarketCap
- Price aggregation and caching in Redis
- Subscription management (per user symbol list)
- Broadcasting service for price updates using asyncio
- Fallback to HTTP polling if WebSocket unavailable
- Rate limiting per connection

Example Implementation:

```python
# routers/ws_prices.py
from fastapi import APIRouter, WebSocket, WebSocketDisconnect, Depends
from typing import Set
import asyncio
import json

router = APIRouter()

class ConnectionManager:
    def __init__(self):
        self.active_connections: dict[str, WebSocket] = {}
        self.user_subscriptions: dict[str, Set[str]] = {}

    async def connect(self, websocket: WebSocket, user_id: str):
        await websocket.accept()
        self.active_connections[user_id] = websocket
        self.user_subscriptions[user_id] = set()

    def disconnect(self, user_id: str):
        if user_id in self.active_connections:
            del self.active_connections[user_id]
        if user_id in self.user_subscriptions:
            del self.user_subscriptions[user_id]

    async def subscribe(self, user_id: str, symbols: list[str]):
        if user_id in self.user_subscriptions:
            self.user_subscriptions[user_id].update(symbols)

    async def broadcast_price_update(self, symbol: str, price_data: dict):
        disconnected_users = []

        for user_id, symbols in self.user_subscriptions.items():
            if symbol in symbols:
                websocket = self.active_connections.get(user_id)
                if websocket:
                    try:
                        await websocket.send_json({
                            "type": "price_update",
                            "symbol": symbol,
                            "data": price_data
                        })
                    except Exception:
                        disconnected_users.append(user_id)

        for user_id in disconnected_users:
            self.disconnect(user_id)

manager = ConnectionManager()

@router.websocket("/ws/prices")
async def websocket_prices(
```

```
    websocket: WebSocket,
    token: str,
    db: Session = Depends(get_db)
):
    # Verify JWT token
    try:
        payload = verify_token_from_query(token)
        user_id = payload["user_id"]
    except Exception:
        await websocket.close(code=1008)
        return

    await manager.connect(websocket, user_id)

    try:
        while True:
            # Receive subscription requests
            data = await websocket.receive_json()

            if data["type"] == "subscribe":
                symbols = data["symbols"]
                await manager.subscribe(user_id, symbols)

                # Send current prices immediately
                for symbol in symbols:
                    price_data = await get_price_from_cache(symbol)
                    if price_data:
                        await websocket.send_json({
                            "type": "price_update",
                            "symbol": symbol,
                            "data": price_data
                        })

    except WebSocketDisconnect:
        manager.disconnect(user_id)

# Background task for fetching prices
async def price_fetcher_task():
    while True:
        # Get all unique symbols from active subscriptions
        all_symbols = set()
        for symbols in manager.user_subscriptions.values():
            all_symbols.update(symbols)

        # Fetch prices from market data API
        if all_symbols:
            prices = await fetch_prices_from_coingecko(list(all_symbols))

            # Update Redis cache and broadcast
            for symbol, price_data in prices.items():
                await cache_price_in_redis(symbol, price_data)
                await manager.broadcast_price_update(symbol, price_data)
```

```
        await asyncio.sleep(10)  # Update every 10 seconds
```

Database (Redis Cache)

```
# Redis key structure
Key: f"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
```

```
# services/market_data.py
import aiohttp
import asyncio
from typing import Dict, List

class MarketDataService:
    def __init__(self, redis_client):
        self.redis = redis_client
        self.coingecko_api = "https://api.coingecko.com/api/v3"
        self.session: aiohttp.ClientSession | None = None

    async def fetch_prices(self, symbols: List[str], currency: str = "usd") -> Dict:
        """Fetch prices from CoinGecko API"""
        if not self.session:
            self.session = aiohttp.ClientSession()

        ids = ",".join(symbols).lower()
        url = f"{self.coingecko_api}/simple/price"
        params = {
            "ids": ids,
            "vs_currencies": currency,
            "include_24hr_change": "true",
            "include_24hr_vol": "true"
        }

        try:
            async with self.session.get(url, params=params) as response:
                if response.status == 200:
                    return await response.json()
                else:
                    # Fallback to cached data
                    return await self.get_cached_prices(symbols, currency)
        except Exception as e:
            print(f"Error fetching prices: {e}")
```

```
            return await self.get_cached_prices(symbols, currency)

    async def cache_price(self, symbol: str, currency: str, data: dict):
        """Cache price data in Redis"""
        key = f"price:{symbol}:{currency}"
        await self.redis.setex(key, 60, json.dumps(data))

    async def get_cached_prices(self, symbols: List[str], currency: str) -> Dict:
        """Retrieve cached prices from Redis"""
        result = {}
        for symbol in symbols:
            key = f"price:{symbol}:{currency}"
            cached = await self.redis.get(key)
            if cached:
                result[symbol] = json.loads(cached)
        return result
```

API Integration

CoinGecko API:

- Authorization: Free tier (50 calls/min)
- Endpoint: `/simple/price`
- Polling interval: 10 seconds for subscribed symbols
- Batch requests for multiple symbols

Fallback: CoinMarketCap API when rate limits exceeded

Testing Requirements

- Unit tests: WebSocket message handling (pytest)
- Unit tests: Price formatting and currency conversion (pytest)
- Component tests: Price display updates (Jest + React Testing Library)
- Integration test: WebSocket connection and updates (pytest + httpx)
- Load test: 1000+ concurrent WebSocket connections (locust)
- 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 FastAPI
- [ ] WebSocket client implemented in React
- [ ] Market data API integration completed (CoinGecko)
- [ ] 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: `routers/ws_prices.py`, `services/market_data.py`
- Infrastructure: Redis cache

Dependencies:

- CoinGecko/CoinMarketCap API keys obtained
- Redis instance configured and accessible
- WebSocket infrastructure deployed
- Currency conversion rates API configured
- `aiohttp` library for async HTTP requests
- `websockets` library

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 (FastAPI + Python)

- `GET /api/dashboard` endpoint with portfolio aggregation
- `GET /api/portfolios/{id}/summary` endpoint
- Portfolio value calculation service
- Asset allocation calculation logic
- Percentage change calculations (24h, 7d, 30d)
- Query optimization for large portfolios
- Response caching for frequently accessed data

Example Implementation:

```python
# routers/dashboard.py
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from sqlalchemy import func
from decimal import Decimal

router = APIRouter(prefix="/api", tags=["dashboard"])

@router.get("/dashboard")
async def get_dashboard(
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
    redis = Depends(get_redis)
):
    # Check cache first
    cache_key = f"dashboard:{current_user.id}"
    cached = await redis.get(cache_key)
    if cached:
        return json.loads(cached)

    # Get all user portfolios
    portfolios = db.query(Portfolio).filter(
        Portfolio.user_id == current_user.id
    ).all()

    total_value = Decimal(0)
    total_change_24h = Decimal(0)
    all_holdings = []

    for portfolio in portfolios:
        portfolio_data = await calculate_portfolio_value(portfolio, db, redis)
        total_value += portfolio_data["total_value"]
        total_change_24h += portfolio_data["change_24h"]
        all_holdings.extend(portfolio_data["holdings"])

    # Calculate asset allocation
    allocation = calculate_asset_allocation(all_holdings, total_value)

    # Get top 5 holdings
    top_holdings = sorted(
        all_holdings,
        key=lambda x: x["current_value"],
        reverse=True
    )[:5]

    result = {
        "total_value": float(total_value),
        "change_24h": float(total_change_24h),
        "percent_change_24h": float((total_change_24h / total_value) * 100) if total_value >
```

```
0 else 0,
        "allocation": allocation,
        "top_holdings": top_holdings,
        "portfolio_count": len(portfolios)
    }

    # Cache for 30 seconds
    await redis.setex(cache_key, 30, json.dumps(result))

    return result

async def calculate_portfolio_value(portfolio: Portfolio, db: Session, redis):
    """Calculate total value and holdings for a portfolio"""
    holdings = db.query(Holding).filter(
        Holding.portfolio_id == portfolio.id
    ).all()

    holdings_data = []
    total_value = Decimal(0)
    total_cost = Decimal(0)

    for holding in holdings:
        # Get current price from cache
        price_key = f"price:{holding.symbol}:usd"
        price_data = await redis.get(price_key)

        if price_data:
            price_info = json.loads(price_data)
            current_price = Decimal(str(price_info["price"]))
        else:
            current_price = Decimal(0)

        current_value = holding.quantity * current_price
        cost_basis = holding.quantity * holding.purchase_price
        unrealized_pnl = current_value - cost_basis

        holdings_data.append({
            "symbol": holding.symbol,
            "quantity": float(holding.quantity),
            "purchase_price": float(holding.purchase_price),
            "current_price": float(current_price),
            "current_value": float(current_value),
            "unrealized_pnl": float(unrealized_pnl),
            "unrealized_pnl_percent": float((unrealized_pnl / cost_basis) * 100) if
cost_basis > 0 else 0
        })

        total_value += current_value
        total_cost += cost_basis

    change_24h = total_value - total_cost
```

```
    return {
        "total_value": total_value,
        "change_24h": change_24h,
        "holdings": holdings_data
    }

def calculate_asset_allocation(holdings: list, total_value: Decimal) -> dict:
    """Calculate percentage allocation for each asset"""
    allocation = {}

    for holding in holdings:
        symbol = holding["symbol"]
        value = Decimal(str(holding["current_value"]))
        percentage = float((value / total_value) * 100) if total_value > 0 else 0

        if symbol in allocation:
            allocation[symbol]["value"] += float(value)
            allocation[symbol]["percentage"] += percentage
        else:
            allocation[symbol] = {
                "value": float(value),
                "percentage": percentage
            }

    return allocation
```

Database Queries (SQLAlchemy)

```
# Optimized query for portfolio summary
from sqlalchemy import select, func

async def get_portfolio_summary_optimized(portfolio_id: UUID, db: Session):
    """Optimized query with joins"""
    query = select(
        Holding.symbol,
        func.sum(Holding.quantity).label('total_quantity'),
        func.avg(Holding.purchase_price).label('avg_purchase_price'),
        func.sum(Holding.quantity * Holding.purchase_price).label('total_cost')
    ).where(
        Holding.portfolio_id == portfolio_id
    ).group_by(Holding.symbol)

    result = await db.execute(query)
    return result.all()
```

Performance Optimization

- Database connection pooling (20 connections via SQLAlchemy)
- Query result caching in Redis (30 second TTL)
- Lazy loading for non-critical components
- Pagination for large holdings lists

- Code splitting for dashboard bundle

Testing Requirements

- Unit tests: Value calculation logic (pytest)
- Unit tests: Percentage calculation accuracy (pytest)
- Component tests: Dashboard components (Jest + React Testing Library)
- Integration test: Full dashboard load and interaction (pytest + httpx)
- Performance test: Dashboard load time with 100 assets (Lighthouse)
- Performance test: 95th percentile load time ≤3s (locust)
- 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: `routers/dashboard.py`, `services/portfolio_aggregator.py`
- 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 (FastAPI + Python)

- `GET /api/exchanges/connect/{provider}` endpoint (initiate OAuth)
- `GET /api/exchanges/callback/{provider}` endpoint (OAuth callback)
- `GET /api/exchanges` endpoint (list connected exchanges)
- `DELETE /api/exchanges/{id}` endpoint (disconnect)
- OAuth 2.0 client implementation using `authlib` for Binance
- OAuth 2.0 client implementation using `authlib` for Coinbase Pro
- API key encryption service using `cryptography` (AES-256-GCM)
- Credential validation before storage
- Read-only scope enforcement
- Audit logging service

Example Implementation:

```
# routers/exchange.py
from fastapi import APIRouter, Depends, HTTPException
from authlib.integrations.starlette_client import OAuth
from starlette.requests import Request
from services.encryption import encrypt_data, decrypt_data

router = APIRouter(prefix="/api/exchanges", tags=["exchanges"])

# OAuth configuration
oauth = OAuth()

oauth.register(
    name='binance',
    client_id=settings.BINANCE_CLIENT_ID,
    client_secret=settings.BINANCE_CLIENT_SECRET,
    authorize_url='https://accounts.binance.com/oauth/authorize',
    access_token_url='https://accounts.binance.com/oauth/token',
    client_kwargs={'scope': 'user:read trade:read'}
)

oauth.register(
    name='coinbase',
    client_id=settings.COINBASE_CLIENT_ID,
    client_secret=settings.COINBASE_CLIENT_SECRET,
    authorize_url='https://www.coinbase.com/oauth/authorize',
```

```
    access_token_url='https://api.coinbase.com/oauth/token',
    client_kwargs={'scope': 'wallet:accounts:read wallet:transactions:read'}
)

@router.get("/connect/{provider}")
async def connect_exchange(
    provider: str,
    request: Request,
    current_user: User = Depends(get_current_user)
):
    """Initiate OAuth flow for exchange connection"""
    if provider not in ['binance', 'coinbase']:
        raise HTTPException(status_code=400, detail="Unsupported exchange")

    redirect_uri = f"{settings.FRONTEND_URL}/exchange/callback/{provider}"

    return await oauth.create_client(provider).authorize_redirect(
        request,
        redirect_uri,
        state=current_user.id  # CSRF protection
    )

@router.get("/callback/{provider}")
async def exchange_callback(
    provider: str,
    request: Request,
    db: Session = Depends(get_db)
):
    """Handle OAuth callback"""
    client = oauth.create_client(provider)

    try:
        # Exchange authorization code for token
        token = await client.authorize_access_token(request)

        # Validate state parameter
        user_id = request.query_params.get('state')

        # Validate API credentials with exchange
        is_valid = await validate_exchange_credentials(provider, token)

        if not is_valid:
            raise HTTPException(status_code=400, detail="Invalid credentials")

        # Encrypt and store credentials
        encrypted_access_token, nonce_at = encrypt_data(token['access_token'])
        encrypted_refresh_token, nonce_rt = encrypt_data(token.get('refresh_token', ''))

        connection = ExchangeConnection(
            user_id=user_id,
            exchange_name=provider,
            oauth_access_token_encrypted=encrypted_access_token,
```

```
            oauth_refresh_token_encrypted=encrypted_refresh_token,
            encryption_nonce=nonce_at,
            scopes=token.get('scope', '').split(),
            connection_status='active'
        )

        db.add(connection)
        db.commit()

        # Audit log
        await log_exchange_connection(user_id, provider, request)

        return {"message": f"{provider.title()} connected successfully"}

    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

@router.get("/")
async def list_exchanges(
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """List all connected exchanges for user"""
    connections = db.query(ExchangeConnection).filter(
        ExchangeConnection.user_id == current_user.id
    ).all()

    return [
        {
            "id": str(conn.id),
            "exchange_name": conn.exchange_name,
            "status": conn.connection_status,
            "last_sync": conn.last_sync_at,
            "connected_at": conn.created_at
        }
        for conn in connections
    ]

@router.delete("/{connection_id}")
async def disconnect_exchange(
    connection_id: UUID,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Disconnect and delete exchange connection"""
    connection = db.query(ExchangeConnection).filter(
        ExchangeConnection.id == connection_id,
        ExchangeConnection.user_id == current_user.id
    ).first()

    if not connection:
        raise HTTPException(status_code=404, detail="Connection not found")
```

```
    # Audit log before deletion
    await log_exchange_disconnection(current_user.id, connection.exchange_name)

    db.delete(connection)
    db.commit()

    return {"message": "Exchange disconnected successfully"}
```

Encryption Service

```
# services/encryption.py
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import os
import base64

# Master encryption key (stored in environment, rotate annually)
MASTER_KEY = base64.b64decode(os.getenv('ENCRYPTION_MASTER_KEY'))

def encrypt_data(plaintext: str) -> tuple[bytes, bytes]:
    """Encrypt data using AES-256-GCM"""
    aesgcm = AESGCM(MASTER_KEY)
    nonce = os.urandom(12)  # 96-bit nonce

    ciphertext = aesgcm.encrypt(
        nonce,
        plaintext.encode('utf-8'),
        None  # No additional authenticated data
    )

    return ciphertext, nonce

def decrypt_data(ciphertext: bytes, nonce: bytes) -> str:
    """Decrypt data using AES-256-GCM"""
    aesgcm = AESGCM(MASTER_KEY)

    plaintext = aesgcm.decrypt(
        nonce,
        ciphertext,
        None
    )

    return plaintext.decode('utf-8')

async def validate_exchange_credentials(provider: str, token: dict) -> bool:
    """Validate credentials with exchange API"""
    # Implementation depends on exchange
    # Make test API call to verify credentials work
    return True
```

Database (PostgreSQL with SQLAlchemy)

```
# db/models/exchange.py
from sqlalchemy import Column, String, ARRAY, DateTime, LargeBinary
from sqlalchemy.dialects.postgresql import UUID
import uuid
from datetime import datetime

class ExchangeConnection(Base):
    __tablename__ = "exchange_connections"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey("users.id", ondelete="CASCADE"),
nullable=False)
    exchange_name = Column(String(50), nullable=False)  # 'binance', 'coinbase'
    api_key_encrypted = Column(LargeBinary)
    api_secret_encrypted = Column(LargeBinary)
    encryption_nonce = Column(LargeBinary, nullable=False)
    oauth_access_token_encrypted = Column(LargeBinary)
    oauth_refresh_token_encrypted = Column(LargeBinary)
    scopes = Column(ARRAY(String))
    connection_status = Column(String(20), default='active')  # 'active', 'invalid',
'expired'
    last_sync_at = Column(DateTime)
    created_at = Column(DateTime, default=datetime.utcnow)
    updated_at = Column(DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    __table_args__ = (
        Index('idx_exchange_connections_user_id', 'user_id'),
    )

class ExchangeConnectionAudit(Base):
    __tablename__ = "exchange_connection_audit"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey("users.id"))
    exchange_connection_id = Column(UUID(as_uuid=True),
ForeignKey("exchange_connections.id"))
    action = Column(String(50), nullable=False)  # 'connected', 'disconnected', 'validated'
    ip_address = Column(String(45))
    user_agent = Column(Text)
    created_at = Column(DateTime, default=datetime.utcnow)
```

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 via `cryptography` library)
- TLS 1.3 in transit
- CSRF protection for OAuth callbacks (state parameter)
- State parameter validation
- Scope restriction (read-only enforced)
- Rate limiting on connection attempts

Testing Requirements

- Unit tests: Encryption/decryption functions (pytest)
- Unit tests: OAuth flow logic (pytest)
- Integration test: Full OAuth connection flow (pytest + httpx)
- 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 (`authlib`)
- [ ] AES-256 encryption service implemented and tested (`cryptography`)
- [ ] 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 (Alembic)
- [ ] 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: `routers/exchange.py`, `services/encryption.py`
- 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
- `authlib` library for OAuth
- `cryptography` library for AES-GCM encryption

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 (FastAPI + Python)

- `PUT /api/user/password` endpoint (change password)
- `POST /api/user/2fa/enable` endpoint
- `DELETE /api/user/2fa/disable` endpoint
- `GET /api/user/sessions` endpoint (list active sessions)
- `DELETE /api/user/sessions/{id}` endpoint (logout specific session)
- `POST /api/user/sessions/logout-all` endpoint
- `GET /api/user/security-log` endpoint (paginated)
- Device fingerprinting using `user-agents` library
- IP geolocation service integration
- Email notification service integration via SendGrid

Example Implementation:

```
# routers/security.py
from fastapi import APIRouter, Depends, HTTPException, BackgroundTasks
from pydantic import BaseModel, constr
from passlib.context import CryptContext
from user_agents import parse
```

```python
router = APIRouter(prefix="/api/user", tags=["security"])
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

class PasswordChange(BaseModel):
    current_password: str
    new_password: constr(min_length=8, regex=r'^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]')
    confirm_password: str

@router.put("/password")
async def change_password(
    data: PasswordChange,
    request: Request,
    background_tasks: BackgroundTasks,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    # Verify passwords match
    if data.new_password != data.confirm_password:
        raise HTTPException(status_code=400, detail="Passwords do not match")

    # Verify current password
    if not pwd_context.verify(data.current_password, current_user.password_hash):
        raise HTTPException(status_code=400, detail="Current password is incorrect")

    # Hash new password
    new_hash = pwd_context.hash(data.new_password)
    current_user.password_hash = new_hash
    current_user.password_changed_at = datetime.utcnow()

    db.commit()

    # Log security event
    await log_security_event(
        user_id=current_user.id,
        event_type="password_changed",
        ip_address=request.client.host,
        user_agent=request.headers.get("user-agent"),
        db=db
    )

    # Send email notification
    background_tasks.add_task(
        send_security_notification,
        current_user.email,
        "password_changed"
    )

    return {"message": "Password updated successfully"}

@router.get("/sessions")
async def list_sessions(
```

```
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """List all active sessions for user"""
    sessions = db.query(Session).filter(
        Session.user_id == current_user.id,
        Session.expires_at > datetime.utcnow()
    ).order_by(Session.last_activity.desc()).all()

    session_list = []
    for session in sessions:
        device_info = parse_device_info(session.device_info)
        location = await get_location_from_ip(session.ip_address)

        session_list.append({
            "id": str(session.id),
            "device": device_info["device"],
            "browser": device_info["browser"],
            "os": device_info["os"],
            "ip_address": session.ip_address,
            "location": location,
            "last_activity": session.last_activity,
            "created_at": session.created_at
        })

    return session_list

@router.delete("/sessions/{session_id}")
async def logout_session(
    session_id: UUID,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
    redis = Depends(get_redis)
):
    """Logout specific session"""
    session = db.query(Session).filter(
        Session.id == session_id,
        Session.user_id == current_user.id
    ).first()

    if not session:
        raise HTTPException(status_code=404, detail="Session not found")

    # Delete from Redis
    await redis.delete(f"session:{session.token_hash}")

    # Delete from database
    db.delete(session)
    db.commit()

    # Log event
    await log_security_event(
```

```
        user_id=current_user.id,
        event_type="session_terminated",
        metadata={"session_id": str(session_id)},
        db=db
    )

    return {"message": "Session terminated successfully"}

@router.post("/sessions/logout-all")
async def logout_all_sessions(
    request: Request,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
    redis = Depends(get_redis)
):
    """Logout all sessions except current one"""
    current_token = request.headers.get("authorization", "").replace("Bearer ", "")
    current_token_hash = hashlib.sha256(current_token.encode()).hexdigest()

    # Get all sessions except current
    sessions = db.query(Session).filter(
        Session.user_id == current_user.id,
        Session.token_hash != current_token_hash
    ).all()

    # Delete from Redis and database
    for session in sessions:
        await redis.delete(f"session:{session.token_hash}")
        db.delete(session)

    db.commit()

    # Log event
    await log_security_event(
        user_id=current_user.id,
        event_type="all_sessions_terminated",
        db=db
    )

    return {"message": f"{len(sessions)} sessions terminated"}

@router.get("/security-log")
async def get_security_log(
    page: int = 1,
    limit: int = 50,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Get paginated security event log"""
    offset = (page - 1) * limit

    events = db.query(SecurityEvent).filter(
```

```
        SecurityEvent.user_id == current_user.id
    ).order_by(SecurityEvent.created_at.desc()).offset(offset).limit(limit).all()

    total = db.query(SecurityEvent).filter(
        SecurityEvent.user_id == current_user.id
    ).count()

    return {
        "events": [
            {
                "event_type": event.event_type,
                "ip_address": event.ip_address,
                "location": event.location,
                "timestamp": event.created_at,
                "metadata": event.metadata
            }
            for event in events
        ],
        "total": total,
        "page": page,
        "pages": (total + limit - 1) // limit
    }

def parse_device_info(user_agent_string: str) -> dict:
    """Parse user agent to extract device information"""
    ua = parse(user_agent_string)

    return {
        "device": ua.device.family,
        "browser": f"{ua.browser.family} {ua.browser.version_string}",
        "os": f"{ua.os.family} {ua.os.version_string}"
    }

async def get_location_from_ip(ip_address: str) -> str:
    """Get location from IP address using geolocation service"""
    # Integration with IP geolocation API (e.g., ipapi.co)
    try:
        async with aiohttp.ClientSession() as session:
            async with session.get(f"https://ipapi.co/{ip_address}/json/") as resp:
                data = await resp.json()
                return f"{data.get('city', 'Unknown')}, {data.get('country_name',
'Unknown')}"
    except:
        return "Unknown"
```

Database (PostgreSQL with SQLAlchemy)

```
# db/models/security_events.py
from sqlalchemy import Column, String, DateTime, JSON
from sqlalchemy.dialects.postgresql import UUID
import uuid
```

```python
from datetime import datetime

class SecurityEvent(Base):
    __tablename__ = "security_events"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    user_id = Column(UUID(as_uuid=True), ForeignKey("users.id", ondelete="CASCADE"),
nullable=False)
    event_type = Column(String(50), nullable=False)
    # 'login_success', 'login_failed', 'password_changed', '2fa_enabled', '2fa_disabled',
'session_terminated'
    ip_address = Column(String(45))
    user_agent = Column(Text)
    location = Column(String(255))  # City, Country from IP geolocation
    metadata = Column(JSON)
    created_at = Column(DateTime, default=datetime.utcnow)

    __table_args__ = (
        Index('idx_security_events_user_id', 'user_id'),
        Index('idx_security_events_created_at', 'created_at'),
    )

# Add to User model
# password_changed_at = Column(DateTime)
# last_security_review_at = Column(DateTime)
```

Email Templates (SendGrid)

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

Testing Requirements

- Unit tests: Password validation and hashing (pytest)
- Component tests: Security settings forms (Jest + React Testing Library)
- Integration test: Password change flow (pytest + httpx)
- Integration test: Session management (pytest)
- 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 (`user-agents`)

- [ ] IP geolocation integration working
- [ ] Email notification templates created (SendGrid)
- [ ] 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: `routers/security.py` , `services/security_log.py`
- Database: `security_events` table

Dependencies:

- Email service configured (SendGrid)
- IP geolocation API configured (ipapi.co or similar)
- `user-agents` library installed for user agent parsing

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 Script (Python + Bash)

```
#!/bin/bash
# backup.sh - Daily database backup script

DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_FILE="crypto_portfolio_$DATE.sql"
ENCRYPTED_FILE="$BACKUP_FILE.enc"

# PostgreSQL dump
pg_dump -h $DB_HOST -U $DB_USER $DB_NAME > $BACKUP_FILE

# Encrypt with AES-256
openssl enc -aes-256-cbc -salt -in $BACKUP_FILE -out $ENCRYPTED_FILE -k $ENCRYPTION_KEY

# Calculate checksum
SHA256=$(sha256sum $ENCRYPTED_FILE | awk '{print $1}')

# Upload to S3
aws s3 cp $ENCRYPTED_FILE s3://$S3_BUCKET/backups/$ENCRYPTED_FILE \
  --metadata checksum=$SHA256,backup_date=$DATE
```

```
# Cleanup local files
rm $BACKUP_FILE $ENCRYPTED_FILE

# Log completion
echo "Backup completed: $ENCRYPTED_FILE (SHA256: $SHA256)" >> /var/log/backups.log
```

Backup Service (Python with Celery)

```
# tasks/backup.py
from celery import Celery
from celery.schedules import crontab
import subprocess
import boto3
import hashlib
from datetime import datetime
from cryptography.fernet import Fernet
import logging

app = Celery('backup_tasks', broker='redis://localhost:6379/0')

# Configure periodic task
app.conf.beat_schedule = {
    'daily-backup': {
        'task': 'tasks.backup.perform_daily_backup',
        'schedule': crontab(hour=2, minute=0),  # Daily at 2:00 AM UTC
    },
}

logger = logging.getLogger(__name__)

@app.task(bind=True, max_retries=3)
def perform_daily_backup(self):
    """Perform daily encrypted database backup"""
    try:
        backup_file = create_postgres_backup()
        encrypted_file = encrypt_backup(backup_file)
        checksum = calculate_checksum(encrypted_file)
        s3_key = upload_to_s3(encrypted_file, checksum)

        # Cleanup local files
        os.remove(backup_file)
        os.remove(encrypted_file)

        # Log success
        logger.info(f"Backup completed: {s3_key} (checksum: {checksum})")

        # Record backup metadata
        record_backup_metadata(s3_key, checksum)

        return {"status": "success", "s3_key": s3_key, "checksum": checksum}
```

```
    except Exception as e:
        logger.error(f"Backup failed: {str(e)}")

        # Send alert
        send_backup_failure_alert(str(e))

        # Retry
        raise self.retry(exc=e, countdown=300)  # Retry after 5 minutes

def create_postgres_backup() -> str:
    """Create PostgreSQL database dump"""
    timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    backup_file = f"/tmp/crypto_portfolio_{timestamp}.sql"

    cmd = [
        "pg_dump",
        "-h", os.getenv("DB_HOST"),
        "-U", os.getenv("DB_USER"),
        "-d", os.getenv("DB_NAME"),
        "-f", backup_file
    ]

    result = subprocess.run(cmd, capture_output=True, text=True)

    if result.returncode != 0:
        raise Exception(f"pg_dump failed: {result.stderr}")

    return backup_file

def encrypt_backup(backup_file: str) -> str:
    """Encrypt backup file using AES-256"""
    encryption_key = os.getenv("BACKUP_ENCRYPTION_KEY").encode()
    fernet = Fernet(encryption_key)

    encrypted_file = f"{backup_file}.enc"

    with open(backup_file, 'rb') as f:
        data = f.read()

    encrypted_data = fernet.encrypt(data)

    with open(encrypted_file, 'wb') as f:
        f.write(encrypted_data)

    return encrypted_file

def calculate_checksum(file_path: str) -> str:
    """Calculate SHA-256 checksum"""
    sha256_hash = hashlib.sha256()

    with open(file_path, "rb") as f:
```

```
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)

    return sha256_hash.hexdigest()

def upload_to_s3(file_path: str, checksum: str) -> str:
    """Upload encrypted backup to S3"""
    s3_client = boto3.client('s3')
    bucket_name = os.getenv("S3_BACKUP_BUCKET")

    timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    s3_key = f"backups/{timestamp}.sql.enc"

    s3_client.upload_file(
        file_path,
        bucket_name,
        s3_key,
        ExtraArgs={
            'Metadata': {
                'checksum': checksum,
                'backup_date': timestamp,
                'encrypted': 'true'
            },
            'ServerSideEncryption': 'AES256'
        }
    )

    return s3_key

def record_backup_metadata(s3_key: str, checksum: str):
    """Record backup metadata in database"""
    from db.session import get_db
    from db.models.backup import BackupRecord

    db = next(get_db())

    backup_record = BackupRecord(
        s3_key=s3_key,
        checksum=checksum,
        size_bytes=os.path.getsize(s3_key),
        status='completed',
        created_at=datetime.utcnow()
    )

    db.add(backup_record)
    db.commit()

def send_backup_failure_alert(error_message: str):
    """Send alert notification on backup failure"""
    # Email notification
    from core.email_utils import send_email
```

```
    send_email(
        to=os.getenv("ADMIN_EMAIL"),
        subject="⚠️ Database Backup Failed",
        body=f"Backup failed at {datetime.utcnow()}\n\nError: {error_message}"
    )

    # Slack notification (optional)
    try:
        import requests
        webhook_url = os.getenv("SLACK_WEBHOOK_URL")
        if webhook_url:
            requests.post(webhook_url, json={
                "text": f"🚨 Database Backup Failed\n```{error_message}```"
            })
    except:
        pass

@app.task
def cleanup_old_backups():
    """Delete backups older than 90 days"""
    s3_client = boto3.client('s3')
    bucket_name = os.getenv("S3_BACKUP_BUCKET")

    # Calculate cutoff date (90 days ago)
    cutoff_date = datetime.utcnow() - timedelta(days=90)

    response = s3_client.list_objects_v2(
        Bucket=bucket_name,
        Prefix='backups/'
    )

    deleted_count = 0

    for obj in response.get('Contents', []):
        last_modified = obj['LastModified'].replace(tzinfo=None)

        if last_modified < cutoff_date:
            s3_client.delete_object(
                Bucket=bucket_name,
                Key=obj['Key']
            )
            deleted_count += 1
            logger.info(f"Deleted old backup: {obj['Key']}")

    return {"deleted_count": deleted_count}
```

Recovery Procedures

```
# tasks/restore.py
def restore_from_backup(s3_key: str, target_db: str = None):
    """Restore database from encrypted backup"""
```

```
    s3_client = boto3.client('s3')
    bucket_name = os.getenv("S3_BACKUP_BUCKET")

    # Download from S3
    local_encrypted = "/tmp/restore.sql.enc"
    s3_client.download_file(bucket_name, s3_key, local_encrypted)

    # Verify checksum
    response = s3_client.head_object(Bucket=bucket_name, Key=s3_key)
    expected_checksum = response['Metadata']['checksum']
    actual_checksum = calculate_checksum(local_encrypted)

    if expected_checksum != actual_checksum:
        raise Exception("Checksum verification failed!")

    # Decrypt
    local_decrypted = decrypt_backup(local_encrypted)

    # Restore to database
    target_db = target_db or os.getenv("DB_NAME")

    cmd = [
        "psql",
        "-h", os.getenv("DB_HOST"),
        "-U", os.getenv("DB_USER"),
        "-d", target_db,
        "-f", local_decrypted
    ]

    result = subprocess.run(cmd, capture_output=True, text=True)

    if result.returncode != 0:
        raise Exception(f"Restore failed: {result.stderr}")

    # Cleanup
    os.remove(local_encrypted)
    os.remove(local_decrypted)

    logger.info(f"Restore completed from {s3_key} to {target_db}")

    return {"status": "success", "restored_from": s3_key}

def decrypt_backup(encrypted_file: str) -> str:
    """Decrypt backup file"""
    encryption_key = os.getenv("BACKUP_ENCRYPTION_KEY").encode()
    fernet = Fernet(encryption_key)

    decrypted_file = encrypted_file.replace('.enc', '')

    with open(encrypted_file, 'rb') as f:
        encrypted_data = f.read()
```

```
    decrypted_data = fernet.decrypt(encrypted_data)

    with open(decrypted_file, 'wb') as f:
        f.write(decrypted_data)

    return decrypted_file
```

Database Model for Backup Tracking

```
# db/models/backup.py
from sqlalchemy import Column, String, BigInteger, DateTime
from sqlalchemy.dialects.postgresql import UUID
import uuid
from datetime import datetime

class BackupRecord(Base):
    __tablename__ = "backup_records"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    s3_key = Column(String(500), nullable=False)
    checksum = Column(String(64), nullable=False)
    size_bytes = Column(BigInteger)
    status = Column(String(20), default='completed')  # 'completed', 'failed'
    error_message = Column(Text)
    created_at = Column(DateTime, default=datetime.utcnow)
```

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 integration (optional)
import boto3

cloudwatch = boto3.client('cloudwatch')

def send_backup_metric(status: str):
    """Send backup status to CloudWatch"""
    cloudwatch.put_metric_data(
        Namespace='CryptoPortfolio/Backups',
        MetricData=[
            {
                'MetricName': 'BackupStatus',
                'Value': 1 if status == 'success' else 0,
```

```
                'Unit': 'Count',
                'Timestamp': datetime.utcnow()
            }
        ]
    )
```

Testing Requirements

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

Definition of Done

- [ ] Backup script/service implemented (Python + Celery)
- [ ] AWS S3 bucket configured with policies
- [ ] Encryption working (AES-256 verified via `cryptography`)
- [ ] Daily cron job scheduled and running (Celery beat)
- [ ] Checksum validation implemented
- [ ] Monitoring and alerting configured
- [ ] Admin notification system working (email + Slack)
- [ ] Recovery documentation complete
- [ ] Full backup-restore tested successfully
- [ ] Monthly DR drill scheduled
- [ ] Retention policy automated (lifecycle or cleanup task)
- [ ] 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` (optional bash wrapper)
- Backend: `tasks/backup.py`, `tasks/restore.py` (Celery tasks)
- Infrastructure: AWS S3, CloudWatch (optional), SNS (optional)

Dependencies:

- AWS account with S3 access
- PostgreSQL backup utilities installed (`pg_dump`, `psql`)
- `cryptography` library for encryption
- CloudWatch monitoring configured (optional)
- Admin email/Slack webhook configured
- Celery + Redis for task scheduling

- `boto3` library for AWS SDK

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 teams

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)

Updated Tech Stack Summary

Frontend

- **React 18 + TypeScript**
- **Material-UI** for components
- **Chart.js** for data visualization
- **Formik** for form validation
- **Axios** for API calls
- **WebSocket** client for real-time updates

Backend (Python Migration)

- **FastAPI** (async web framework)
- **SQLAlchemy 2.0** (ORM with async support)
- **Alembic** (database migrations)
- **PyJWT** (JWT tokens)
- **passlib + bcrypt** (password hashing)
- **pyotp** (TOTP/2FA)
- **cryptography** (AES-256 encryption)
- **authlib** (OAuth 2.0)
- **Celery + Redis** (background tasks)
- **sendgrid-python** (email service)
- **aiohttp** (async HTTP client)
- **user-agents** (device fingerprinting)
- **slowapi** (rate limiting)

Database

- **PostgreSQL 15** (primary database)

- **Redis 7** (session cache, price cache)

Testing

- **pytest** (unit & integration tests)
- **httpx** (async HTTP testing)
- **locust** (load testing)
- **Jest** (frontend unit tests)
- **React Testing Library** (component tests)
- **Cypress** (E2E tests - if applicable)
- **Postman** (API testing)
- **OWASP ZAP** (security testing)

Infrastructure

- **Docker** (containerization)
- **Uvicorn** (ASGI server)
- **Gunicorn** (process manager)
- **AWS S3** (backup storage)
- **CloudWatch** (monitoring - optional)

Testing Strategy

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

Definition of Done (Sprint Level)

- [ ] All 10 user stories completed and accepted
- [ ] 95%+ code coverage achieved (pytest for backend, Jest for frontend)
- [ ] All high-priority test cases passed
- [ ] Security audit completed with no critical issues
- [ ] Performance requirements met (<3s dashboard load, <10s WebSocket updates)
- [ ] API documentation updated (Postman collections + OpenAPI/Swagger)
- [ ] Code reviewed and merged to develop branch
- [ ] Sprint demo prepared for stakeholders
- [ ] Database migrations created and tested (Alembic)

Risks & Mitigation

1. **Python Migration Complexity**
    - Mitigation: Start with simpler endpoints, use established patterns, reference FastAPI docs
2. **WebSocket Scalability**
    - Mitigation: Load testing Day 11, Redis optimization, consider WebSocket clustering
3. **Security Testing Delays**

    - Mitigation: Parallel security testing throughout, dedicated security review sessions
4. **Database Performance**
    - Mitigation: Query optimization, SQLAlchemy connection pooling, Redis caching
5. **OAuth Integration Challenges**
    - Mitigation: Start early (Day 8), use sandbox environments, thorough testing

Middleware Adjustments

- **CORS middleware** for React frontend (FastAPI CORSMiddleware)
- **Rate limiting** via `slowapi`
- **Session management** with Redis (`aioredis`)
- **JWT auth middleware** (FastAPI dependencies)
- **Request logging** and exception handling (FastAPI middleware)

Migration Complete! 🎉

All user stories have been updated from **Rust + Axum** to **Python + FastAPI** while maintaining:

- ✅ All functional requirements
- ✅ Security standards
- ✅ Performance targets
- ✅ Testing requirements
- ✅ Database schemas (PostgreSQL compatible)
- ✅ Frontend compatibility (React remains unchanged)

Next Steps: Generate Python FastAPI backend scaffold with all boilerplate and mapped routes ready to run!