🚀 COMPLETE PORTFOLIO DASHBOARD - BACKEND DEVELOPMENT PLAN

📋 DATABASE SCHEMA - FINAL (NO MORE CHANGES)

Your Prisma schema is **LOCKED** and production-ready:[1]

```
✅ User → Stock (One-to-Many)
✅ Stock → PriceData (One-to-Many)
✅ PriceCache (Global symbol cache)
```

No further schema modifications allowed.

🏗 MODULE STRUCTURE & IMPLEMENTATION ORDER

PHASE 1: Stock Management (Priority 1)

Module: `stock/`

Purpose: CRUD operations for user's stock holdings[2] [1]

Files to Create:

```
src/stock/
├── stock.module.ts
├── stock.service.ts
├── stock.controller.ts
├── dto/
│   ├── create-stock.dto.ts
│   ├── update-stock.dto.ts
│   └── bulk-import-stock.dto.ts
└── entities/
    └── stock.entity.ts
```

stock.module.ts

What it does: Registers Stock module dependencies

```
- Imports: PrismaModule, StockPriceModule
- Providers: StockService
- Controllers: StockController
- Exports: StockService
```

stock.service.ts

What it does: Business logic for stock operations

Functions:

Function	Purpose	Parameters	Returns
`createStock()`	Add new stock to user's portfolio	`userId, CreateStockDto`	Created stock object
`getUserStocks()`	Get all stocks for a user	`userId`	Array of stocks
`getStockById()`	Get single stock details	`userId, stockId`	Stock object
`updateStock()`	Update stock details	`userId, stockId, UpdateStockDto`	Updated stock
`deleteStock()`	Remove stock from portfolio	`userId, stockId`	Success message
`bulkCreateStocks()`	Import multiple stocks (Excel data)	`userId, BulkImportDto[]`	Created stocks array
`calculateInvestment()`	Helper to compute investment	`purchasePrice, quantity`	`purchasePrice * quantity`
`calculatePortfolioPercent()`	Calculate stock's % in portfolio	`investment, totalInvestment`	Percentage value
`getStocksBySymbol()`	Find stocks by symbol	`userId, symbol`	Array of matching stocks
`getStocksBySector()`	Get stocks in specific sector	`userId, sector`	Array of stocks

Business Rules:

- Validate unique stock symbol per user (enforced by DB constraint)[1]
- Auto-calculate `investment = purchasePrice * quantity`
- Auto-calculate `portfolioPercent` after fetch

- Throw `ConflictException` if duplicate symbol added

󰊠 stock.controller.ts

What it does: HTTP endpoints for stock operations[3]

Endpoints:

Method	Endpoint	Description	Request Body	Response
POST	`/api/stocks`	Add new stock	`CreateStockDto`	Created stock
GET	`/api/stocks`	Get all user stocks	-	Array of stocks
GET	`/api/stocks/:id`	Get single stock	-	Stock details
PATCH	`/api/stocks/:id`	Update stock	`UpdateStockDto`	Updated stock
DELETE	`/api/stocks/:id`	Delete stock	-	Success message
POST	`/api/stocks/bulk`	Import multiple stocks	`BulkImportDto[]`	Created stocks
GET	`/api/stocks/sector/:sector`	Filter by sector	-	Sector stocks
GET	`/api/stocks/search`	Search by symbol/name	`?query=HDFC`	Matching stocks

Auth: All endpoints protected by `AccessTokenGuard` (already implemented)[1]

󰊠 dto/create-stock.dto.ts

What it does: Validates incoming stock data

```
Required Fields:
- symbol: string (e.g., "HDFCBANK.NS")
- name: string (e.g., "HDFC Bank")
- sector: string (e.g., "Financial Sector")
- purchasePrice: number
- quantity: number
- exchange: string ("NSE" or "BSE")

Optional Fields:
- marketCap, peRatioTTM, latestEarnings, revenueTTM
- ebitdaTTM, ebitdaPercent, pat, patPercent
- All growth metrics, ratios, etc.

Auto-Calculated (not in DTO):
```

```
- investment (server calculates)
- portfolioPercent (calculated during fetch)
```

Validation:

- `@IsNotEmpty()` for required fields
- `@IsPositive()` for prices/quantities
- `@IsEnum()` for exchange (NSE/BSE)

dto/update-stock.dto.ts

What it does: Allows partial updates

```
All fields optional (extends PartialType(CreateStockDto))
- Can update purchasePrice, quantity
- Can update fundamental data
- Investment auto-recalculates if price/qty changes
```

dto/bulk-import-stock.dto.ts

What it does: Import from Excel file data[2]

```
Array of CreateStockDto objects
- Validates entire array
- Skips duplicates (by symbol)
- Returns success/failed count
```

PHASE 2: Stock Price Fetching (Priority 2)

Module: `stock-price/`

Purpose: Fetch real-time market data from external APIs[3]

Files to Create:

```
src/stock-price/
├── stock-price.module.ts
├── stock-price.service.ts
├── stock-price.controller.ts
└── interfaces/
    ├── stock-price-response.interface.ts
    └── yahoo-finance-response.interface.ts
```

stock-price.service.ts

What it does: Fetches market data from Yahoo Finance & Google Finance[3]

Functions:

Function	Purpose	Parameters	Returns
`fetchCurrentPrice()`	Get CMP from Yahoo Finance	`symbol`	Current price
`fetchFundamentals()`	Get P/E, market cap	`symbol`	Fundamental data object
`fetchBatchPrices()`	Fetch multiple stocks at once	`symbols[]`	Array of prices
`getOrFetchPrice()`	Check cache first, then fetch	`symbol`	Price data
`parseYahooFinance()`	Extract data from Yahoo HTML	`html`	Parsed data
`parseGoogleFinance()`	Extract P/E from Google	`html`	P/E ratio

External API Strategy:[3]

1. **Yahoo Finance:** Use `yahoo-finance2` npm package (unofficial but stable)[1]
2. **Google Finance:** Web scraping with `cheerio` package[1]
3. **Rate Limiting:** Max 5 requests/second
4. **Caching:** Use `PriceCache` table (15-second TTL)

Error Handling:

- Return cached data if API fails
- Log failures to console
- Set price to `null` if unavailable

stock-price.controller.ts

Endpoints:

Method	Endpoint	Description	Response
GET	`/api/price/:symbol`	Get current price	`{symbol, currentPrice, peRatio, timestamp}`
POST	`/api/price/batch`	Fetch multiple prices	Array of price objects
DELETE	`/api/price/cache/:symbol`	Invalidate cache	Success message

PHASE 3: Price Data Tracking (Priority 3)

Module: `price-data/`

Purpose: Store historical price snapshots and calculate gains[2]

Files:

```
src/price-data/
├── price-data.module.ts
├── price-data.service.ts
└── price-data.controller.ts
```

price-data.service.ts

Functions:

Function	Purpose	Parameters	Returns
`createPriceData()`	Store price snapshot	`stockId, priceInfo`	Created PriceData
`getLatestPriceData()`	Get most recent price	`stockId`	Latest PriceData
`getPriceHistory()`	Get historical data	`stockId, startDate, endDate`	Array of PriceData
`calculateGainLoss()`	Compute profit/loss	`purchasePrice, quantity, currentPrice`	`{presentValue, gainLoss, gainLossPercent}`
`updateStockPriceData()`	Fetch fresh price & save	`stockId`	Updated PriceData
`bulkUpdatePrices()`	Update multiple stocks	`stockIds[]`	Updated count
`deleteOldPriceData()`	Cleanup old history	`daysToKeep`	Deleted count

Business Logic:[2]

- `presentValue = currentPrice * quantity`
- `gainLoss = presentValue - investment`
- `gainLossPercent = (gainLoss / investment) * 100`

price-data.controller.ts

Endpoints:

Method	Endpoint	Description
GET	`/api/price-data/:stockId/latest`	Get most recent price
GET	`/api/price-data/:stockId/history`	Get historical prices
POST	`/api/price-data/:stockId/refresh`	Force price update

PHASE 4: Portfolio Analytics (Priority 4)

Module: `portfolio/`

Purpose: Aggregate metrics across all user stocks[2]

Files:

```
src/portfolio/
├── portfolio.module.ts
├── portfolio.service.ts
├── portfolio.controller.ts
└── dto/
    └── portfolio-metrics.dto.ts
```

portfolio.service.ts

Functions:

Function	Purpose	Returns
`getUserPortfolio()`	Get all stocks with latest prices	Portfolio with stocks array
`calculateTotalInvestment()`	Sum of all investments	Total investment amount
`calculateCurrentValue()`	Sum of all present values	Current portfolio value
`calculateTotalGainLoss()`	Overall profit/loss	`{totalGainLoss, percentage}`
`getSectorBreakdown()`	Group by sector	Array of `{sector, value, percent}`
`getTopPerformers()`	Stocks with highest gains	Top N stocks
`getUnderperformers()`	Stocks with losses	Bottom N stocks
`getPortfolioMetrics()`	Complete dashboard data	Full metrics object

Aggregation Logic:[2]

- Sum investments across all stocks
- Sum present values
- Calculate sector-wise distribution
- Sort by gain/loss percent

portfolio.controller.ts

Endpoints:

Method	Endpoint	Description	Response Structure
GET	`/api/portfolio`	Complete portfolio	Stocks array with latest prices
GET	`/api/portfolio/metrics`	Dashboard metrics	`{totalInvestment, currentValue, gainLoss, stockCount}`
GET	`/api/portfolio/sector-breakdown`	Sector distribution	Array of sector stats
GET	`/api/portfolio/top-performers`	Best stocks	Top 5 gainers
GET	`/api/portfolio/underperformers`	Worst stocks	Top 5 losers

PHASE 5: Cron Jobs (Priority 5)

Module: `cron/`

Purpose: Automated price updates every 15 seconds[3]

Files:

```
src/cron/
├── cron.module.ts
└── cron.service.ts
```

cron.service.ts

What it does: Background jobs for price updates

Cron Jobs:

Job	Schedule	Purpose	What It Does
`updateAllStockPrices()`	`*/15 * * * * *` (Every 15 sec)	Update all stock prices	1. Get all unique stock symbols 2. Fetch current prices via `StockPriceService` 3. Create `PriceData` entries 4. Log: "Updated X stocks"
`refreshPriceCache()`	`0 */5 * * * *` (Every 5 min)	Clean stale cache	1. Delete expired entries from `PriceCache` 2. Re-fetch top 20 most-viewed stocks
`cleanupOldPriceData()`	`0 0 0 * * *` (Daily midnight)	Delete old history	Delete `PriceData` older than 90 days
`marketOpenUpdate()`	`0 30 9 * * 1-5` (9:30 AM IST Mon-Fri)	Market open hook	1. Invalidate all cache 2. Force fresh data fetch 3. Emit WebSocket event (if implemented)
`marketCloseSnapshot()`	`0 30 15 * * 1-5` (3:30 PM IST)	Market close hook	1. Create final price snapshot 2. Calculate daily portfolio performance

Implementation:

- Use `@nestjs/schedule` package (already installed)[1]
- Import `ScheduleModule.forRoot()` in `AppModule`
- Use `@Cron()` decorator for each job

PHASE 6: Cache Management (Priority 2 - Parallel with Stock Price)

Module: `cache/`

Purpose: Manage `PriceCache` table operations

Files:

```
src/cache/
├── cache.module.ts
└── cache.service.ts
```

cache.service.ts

Functions:

Function	Purpose	Parameters	Returns
`get()`	Retrieve from cache	`symbol`	Cached price or `null`

Function	Purpose	Parameters	Returns
`set()`	Store in cache	`symbol, priceData, ttl`	Created cache entry
`isValid()`	Check if cache expired	`expiresAt`	Boolean
`invalidate()`	Delete cache entry	`symbol`	Success status
`invalidateAll()`	Clear entire cache	-	Deleted count
`batchGet()`	Get multiple cached prices	`symbols[]`	Array of cache entries
`getCacheStats()`	Cache metrics	-	`{total, expired, hitRate}`

Cache Strategy:

- TTL: 15 seconds for price data[3]
- Auto-delete expired entries
- Fallback to API if cache miss

PHASE 7: WebSocket Gateway (Optional Enhancement)

Module: `websocket/`

Purpose: Push real-time price updates to frontend[3]

Files:

```
src/websocket/
├── websocket.module.ts
└── stock.gateway.ts
```

󰅩 stock.gateway.ts

What it does: Real-time communication with frontend

Events:

Event	Direction	Purpose	Payload
`subscribe-stocks`	Client → Server	User subscribes to their stocks	`{stockIds: []}`
`price-update`	Server → Client	Push price changes	`{stockId, currentPrice, gainLoss}`
`market-status`	Server → Client	Market open/close notification	`{status: 'open'/'closed'}`

Implementation:

- Use `@nestjs/websockets` (already installed) [1]
- Emit updates when cron job updates prices
- Auth: Validate JWT token on connection

▯ COMPLETE ENDPOINT REFERENCE

Auth Endpoints ✅ (Already Implemented) [1]

Method	Endpoint	Purpose
POST	`/api/auth/register`	Register new user
POST	`/api/auth/login`	Login and get JWT token

Stock Endpoints

Method	Endpoint	Purpose	Auth Required
POST	`/api/stocks`	Add new stock	✅
GET	`/api/stocks`	Get all user stocks	✅
GET	`/api/stocks/:id`	Get single stock	✅
PATCH	`/api/stocks/:id`	Update stock	✅
DELETE	`/api/stocks/:id`	Delete stock	✅
POST	`/api/stocks/bulk`	Import multiple stocks	✅
GET	`/api/stocks/sector/:sector`	Filter by sector	✅
GET	`/api/stocks/search?query=HDFC`	Search stocks	✅

Stock Price Endpoints

Method	Endpoint	Purpose	Auth Required
GET	`/api/price/:symbol`	Get current price	✅
POST	`/api/price/batch`	Fetch multiple prices	✅
DELETE	`/api/price/cache/:symbol`	Clear cache	✅

Price Data Endpoints

Method	Endpoint	Purpose	Auth Required
GET	`/api/price-data/:stockId/latest`	Latest price	✅
GET	`/api/price-data/:stockId/history`	Historical prices	✅
POST	`/api/price-data/:stockId/refresh`	Force update	✅

Portfolio Endpoints

Method	Endpoint	Purpose	Auth Required
GET	`/api/portfolio`	Complete portfolio	✅
GET	`/api/portfolio/metrics`	Dashboard metrics	✅
GET	`/api/portfolio/sector-breakdown`	Sector stats	✅
GET	`/api/portfolio/top-performers`	Top gainers	✅
GET	`/api/portfolio/underperformers`	Top losers	✅

REQUIRED NPM PACKAGES (Already Installed)[1]

```
✅ @nestjs/schedule - Cron jobs
✅ yahoo-finance2 - Yahoo Finance API
✅ cheerio - Web scraping
✅ axios - HTTP requests
✅ @nestjs/websockets - Real-time updates
✅ socket.io - WebSocket transport
```

IMPLEMENTATION CHECKLIST

Week 1: Core Features

- [ ] ✅ Stock Module (CRUD)
- [ ] ✅ Stock Price Module (API fetching)
- [ ] ✅ Cache Module (PriceCache operations)
- [ ] ✅ Price Data Module (Historical tracking)

Week 2: Advanced Features

- [ ] ✅ Portfolio Module (Aggregations)
- [ ] ✅ Cron Jobs (Auto-updates)
- [ ] ✅ Bulk Import (Excel data)

Week 3: Enhancements

- [ ] ▯ WebSocket Gateway (Real-time)
- [ ] ▯ Dashboard metrics optimization
- [ ] ▯ Unit tests

▯ START HERE - DAY 1 TASKS

1. **Create Stock Module**
   - Create files: `stock.module.ts`, `stock.service.ts`, `stock.controller.ts`
   - Implement `createStock()`, `getUserStocks()`, `deleteStock()`
   - Test endpoints with Postman
2. **Create DTOs**
   - `create-stock.dto.ts` with validation
   - `update-stock.dto.ts`
3. **Test with Demo User**[1]
   - Login as `demo` user (already seeded)
   - Add HDFC Bank stock
   - Verify data in Prisma Studio

▯ NOTES

- **No DB changes allowed** - Schema is finalized[1]
- **Auth already working** - Focus on business logic[1]
- **Use existing packages** - Yahoo Finance, Cheerio already installed[1]
- **15-second updates** - As per requirement[3]
- **Excel data structure** - Match columns from provided spreadsheet[2]

Ready to code! Start with Stock Module.[3] [1]

⁂

1. paste.txt
2. 4E91F82E.xlsx

3. 1428C2DC.pdf