1. Design Overview

This software uses ‘Wt’ a tool combining Web UI with database ORM and REST capabilities, a lightweight database ‘Sqlite’ and parts of ‘Casablanca C++ REST SDK’ to build a stock trading REST interface and Web UI.

With using ‘Wt’ for Web UI, it is convenient for simple web interfaces and uses plain C++. It also helps prevent CSS and CSRF attacks in a natural manner by never having external untrusted source generate any java script for the browser to run. Each user session creates a separate application context on server that in some cases can even be separate OS process, thus affording kernel level isolation for end users.

This application is made up of these components,

1. REST API classes and their validations.
2. Common Controller
3. Object relational mapping classes
4. Web UI classes, Container widgets, Panels.
5. Common utilities, Logging etc.

Except for the Web UI application class (that is responsible for user session and for which the infrastructure assures separate object for each user session), no other object state is maintained in memory. They are all created on stack in response to requests and are stateless. No global or static data is used (unless that is declared a constant expression).

Thus they are good candidates for creating deployments that require service replication. Once replication is easily supported, high availability and scalability is achieved through load balancing and fail-safety features.

Component View

The software supports operations through REST APIs as well as Web UI.

Common controller is utilized for both the operational modes.

The business objects are mapped to each database table but there are additional business objects that do not have a direct mapping.

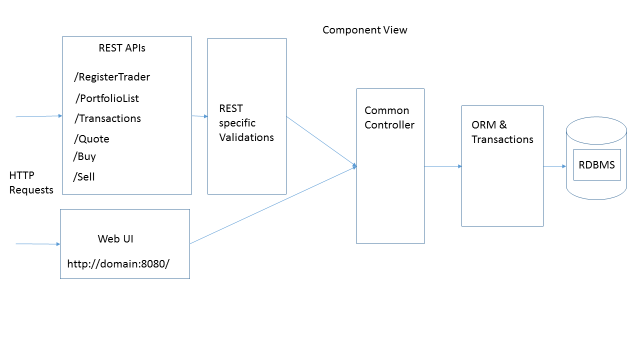
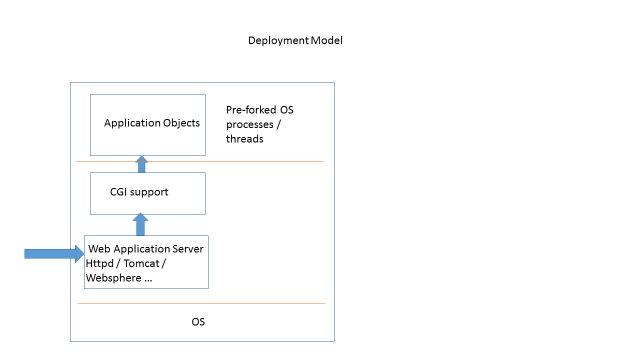
Fig 1. Component View.

Fig 2. Deployment View



Deployment View

If Fast CGI is used, the number of processes / thread resources as well as server memory consumed by the application can usually be controlled by web server configuration. This is essential to prevent Denial of service attacks.

1. Implementation Model

Algorithms:

The important algorithmic elements of the package are,

* Credential validation

Each REST API is accepting a username-password pair for being a stateless server. Credential validation with the persistence for each API invocation is essential for such policy. Alternatively, the browser can send out cookies/tokens but even those will need some session persistence.

* Buy-Sell duality

The ‘Buy’ implementation goes through the following steps after credentials are validated,

1. Check if the stock code is valid.
2. Check if the trader has balance cash given the quantity and offer price.
3. If ‘a’ & ‘b’ pass, buy transaction can proceed and start, otherwise ends with error.
4. Deduct and update balance cash with trader.
5. Update last sale price for the stock.
6. Add to trader transactions.
7. Check trader portfolio for this stock.
8. If this stock already exists, then update quantity and add to total cost.
9. If not, then add a new portfolio entry and update total cost and quantity.
10. End transaction.

The ‘Sell’ is a dual function to ‘Buy’, it differs in that the Portfolio is checked for stock quantity. Also, if the portfolio for this stock reaches zero, then the portfolio entry is deleted.

1. Check if the stock code is valid.
2. Check if the trader has balance stock quantity in portfolio.
3. If ‘a’ & ‘b’ pass, sell transaction can proceed and start, otherwise ends with error.
4. Add and update balance cash with trader.
5. Update last sale price for the stock.
6. Add to trader transactions.
7. Check trader portfolio for this stock.
8. Deduct sold quantity from portfolio entry for the stock.
9. For total cost, calculate the old average price of the stock and write fresh total cost as the remaining quantity multiplied by old average stock price.
10. If quantity of the stock is zero then delete portfolio entry.
11. End transaction.

* For portfolio listing, the current value of each stock entry is calculated using the last sale price of this stock.

REST API signatures:

For each REST API, a separate class is defined that inherits from ‘Wt::WResource’.

‘Wt::WResource’ contains a method ‘handleRequest’ that receives a const HTTP request and an HTTP response by non-const reference.

Wt::WResource::handleRequest(const Request &request, Response &response);

This method is overridden for each REST API class.

Common Controller:

The REST interface and the Web UI share a common controller class ‘CommonController’. This controller is responsible for some validations regarding credentials and presenting an interface to the REST and UI classes. It then delegates the inputs to appropriate persistence layers.

Persistence - Object Relational Mapping (ORM):

One class for each table is created that maps its attributes to the table definition. Each such class has a method specifically for the REST operation to perform.

Each such method is self-contained in terms of acquiring database transaction and ending them.

This, tables named ‘Trader’, ‘Stock’, ‘Portfolio’ and ‘Stock\_Transaction’ have their corresponding object classes mapped.

Two other operations ‘Buy’ and ‘Sell’ correspond to API’s but do not have corresponding tables. They are specially handled since they have multiple table reads and writes. But they create their own transactions and perform persistence operations.

1. Data Model

The third party database engine ‘SQLite 3’ is being used as the relational database. The file named ‘Market.db’ with some seed data is assumed to be present in the project directory.

Optimistic concurrency control

The Object relational mapping support from ‘Wt 3.3.6’ requires using two additional columns with each table, namely ‘id’ and ‘version’. The ‘id’ column is used as an auto-increment column and as the primary key. The ‘version’ column helps maintain versioning on each row of the table and thus enables use of optimistic concurrency model in ‘Wt’ and leads to higher scalability. It happens that the ‘version’ column is incremented on each committed update and the concurrent reads check their version number with that in the database. If the version does not match, each concurrent access can easily re-read the fresh commits. Thus the concurrency control relies on detection rather than avoidance.

This software uses the following persistence schema,

1. Trader-

id integer primary key auto-increment,

--- added for Wt use

version integer not null,

--- added for optimistic policy concurrent accesses

username varchar(100) NOT NULL,

--- input lengths controlled through validations on REST APIs

password varchar(20) NOT NULL,

--- Should be one-way hashed for persistence.

balancecash numeric NOT NULL

1. Stock-

id integer primary key autoincrement,

version integer not null,

stockcode varchar(10),

lastsaleprice numeric NOT NULL

1. Stock\_Transaction-

id integer primary key autoincrement,

version integer not null,

username varchar(100),

stockcode varchar(10),

quantity number NOT NULL,

totalcost numeric NOT NULL

1. Portfolio-

id integer primary key autoincrement,

version integer not null,

username varchar(100),

stockcode varchar(10),

quantity number NOT NULL,

datetime numeric NOT NULL,

status varchar(10) NOT NULL

As seed values, one guest user and 3 individual stocks with their initial prices are added to the schema.