**Architecture**

*A*ll the components are deployed as **Microservices**.

**Microservice Components**

1. **FrontOffice.APIGateway** – **[Authentication, Load-balancing, Aggregation] [Not part of implementation]**
   * All the requests from Front-office apps come here first.
   * Front-office apps supply tokens generated by FrontOffice.Authentication service.
   * Token is validated by contacting authentication service.
   * Calls appropriate service API if the validations is successful.
   * Responsible for load balancing as well.
2. **BackOffice.APIGateway** – **[Authentication, Load-balancing, Aggregation] [Not part of implementation]**
3. **Transaction – [Records all the transactions]**
   * Design ensures any type of database can be plugged-in.
   * Uses entity framework. [Not fully implemented]
   * Dependency Injection. [Not fully implemented]
   * Considering Cassandra DB for transaction recording.[Not implemented]
   * Transactions are recorded to Kafka topics.
   * Kafka topics will be subscribed by apps for real time trade updates.

1. **TradeAnalytics – [Provides generic API for reporting]** 
   * Reports will be cached.
   * If report is not available in the cache only then DB will be contacted.
2. **FrontOffice.Authentication** – **[JWT based authentication]** [Not part of implementation]
3. **BackOffice.Authentication** – **[JWT based authentication]** [Not part of implementation]

**Assumptions**:-

* Testing of the APIs have been performed using Console App(TestWebAPI).
* Real DB has not been integrated, in memory cache has been used to validate the API work-flow.
* Considering Cassandra for transaction recording.
* For real time transactions update to the applications Kafka has been considered.
* Dependency Injection is not implemented, but Dependency objects are injected from Test project.
* Authentication has not been implemented but architecture for authentication has been considered and JWT based authentication has been considered for easy onboarding.
* Report query parameter is a json string but dummy not being used for report querying.
* Currency conversion is not part of the implementation.
* To support scalability and multiple geographies, replicas of microservices will have to be deployed.
* Builder has been considered for report generation although not fully integrated.

Any combination of inputs for reports such as hourly+monthy or weekly + monthly or hourly + weekly + monthly or any custom report can be built and returned as a single object using ReportBuilder.

Report builing examples:-

***Report weekMonth = new ReportBuilder().LastWeek(reportEntityLastWeek).LastMonth(reportEntityLastMonth).build();***

***Report all = new ReportBuilder().LastHour(reportEntityLastHour).LastWeek(reportEntityLastWeek).LastMonth(reportEntityLastMonth).build();***

* Communication between microservices is out of the scope of this solution.
* Error handling and validations are not part of the solution.

**Logger**

* Logger library has been tested using test project(LogTester).
* Logger is a multithreaded logger which uses ConcurrentQueue for recording the messages.
* Log write(to File/DB/UI) is performed asynchronously without engaging the main thread.
* Log extraction is shown in console.
* Observer pattern has been used for subscribing the log consumers, e.g FileLoggerSubscriber, DBLoggerSubscriber, ConsoleLoggerSubscriber.
* These subscribers perform asynchronously.
* Basic logging types Message/Error/Debug has been implemented.
* Logger has a config file for various settings and policies, showing some sample policies but is not being used in the code.
* The snapshot below shows log collection and extraction can happen at the same time without hanging the main UI.

