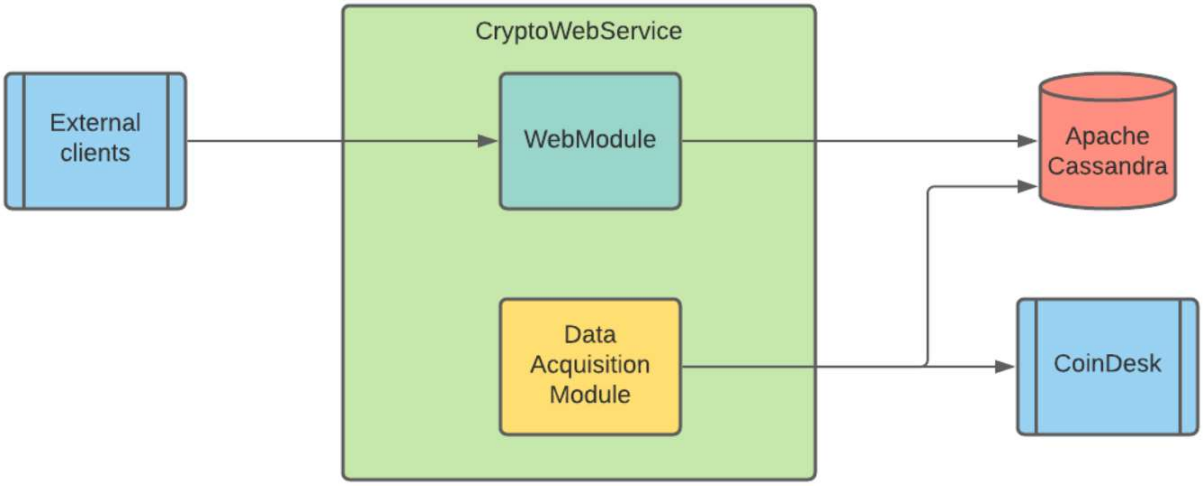
CryptoWebService

1. Architecture



From the architecture point of view, based on the requirements, the following logical modules are identified:

- **Data Acquisition Module** – responsible for polling an external service and getting the Bitcoin to USD rate. Data needs to be stored in storage facility, Apache Cassandra in this case;
- **WebModule** – responsible for exposing data to external clients.

2. Design choices

2.1 Storage

I consider Apache Cassandra to be an optimal choice in terms of storage because of the following:

- Highly scalable and highly available with no single point of failure;
- Horizontal scalling;
- Very high throughput for both write & read;
- Data consistency.

As the requirement was to store time series data basically other suitable choices could have been time series databases like Apache Druid, InfluxDB or Riak. For the first version of the system traditional SQL databases will be suited as well.

However, I assumed that there is a roadmap for the service and that will involve storing the exchanges rates for many other currencies (both fiat & crypto) and serving lots of external clients (UIs, other services from the organization, clients from outside the organization etc.) hence I consider Apache Cassandra a better choice.

The other important part is the partition & clustering of data and the table structure that I chose is below (again having in mind that the service will need to scale & support a lot more currencies):

```
CREATE TABLE crypto.exchange_rate (
  from_currency text,
  to_currency text,
  exchange_date timestamp,
  rate double,
  PRIMARY KEY ((from_currency, to_currency), exchange_date)
  );
```

The currencies compose the partition key, while the date is the clustering key. Because of Cassandra's internal storage format (SSTABLE, sorted strings table) that will allow serving efficiently, without reading too much the request mentioned in the requirement (latest rate, historical rates).

As the partition key is done on currencies, when multiple currencies will be available, the system will scale on write as well - the load will be distributed on multiple nodes.

2.2 Data Acquisition

As the data acquisition module will need to scale a good choice would be the Spring WebFlux's WebClient which allows to make calls to an external service in a non-blocking fashion.

Also, the calls are done using a thread pool mechanism in order to allow parallel execution (may be needed in the future when service will need to support multiple currencies).