Asset management Challenge - Production readiness document

- Stability
  - Need to have stable development process
  - Need to have stable deployment process
  - Need to have automated CI\CD pipeline for faster delivery of stable changes to Production.
  - Build at least following steps in the CI\CD pipeline
    - Build per checkin.
    - Unit testing
    - Integration testing
    - Load testing
    - Quality check
    - Automated deployment to staging
    - Additional steps can be
      - Migration of build from staging to other environment based on process. Example can be approval based.

- Scalability
  - Need to identify the amount of traffic based on which scalability need to be determined.
  - Current system is difficult to scale horizontally as the data is in-memory which when scaled will not provide consolidated and consistent view of data.
  - In current form, the service can be vertically scaled.
  - To scale this service horizontally, the data need to save into database.
  - To scale the system, service will be wrapped in Docker container.
  - Kubernetes can be used to create cluster of service.
  - Configuration can be externalized from the service.

- Fault tolerance
  - Circuit breakers for service
  - Integration testing need to be run on every deployment to ensure different part of system are working fine together.
  - Load testing need to be run to check if the service is not breaking under prescribed load.
  - Implementing chaos testing to test the overall system failure tolerance.
  - As of now, the current system is designed to handle two responsibilities:
    - Account creation
    - Transaction between account
  - Failure in any one responsibility will result in failure of other functionality. So ideally, they need to be modeled as separate services.
  - Account service need to expose endpoint related to creation, updating, deletion, validation of Accounts.
  - Transaction service need to expose endpoint related to transaction execution for the accounts. This service might communicate with Account service related to validation of account.

- Monitoring
  - Proper logging need to be done in addition
    - Correlation id need to be generated for each request to track the request
  - Spring based actuator need to be turned on for tracking key metrics like
    - average response time
    - response

        - status of API endpoints
        - endpoint success rate
    - When using Spring boot 2.0, we can Micrometer which can be used with various other monitoring tools like Prometheus

- Documentation
    - Create architecture diagram of the service and surrounding component. This will help us to visualizing the new feature on where to fit.
    - Create Microservice API's using Open API specification. This json documentation can then be rendered using various tools\library i.e. Swagger.
    - Microservice API can also be documented using SpringFox swagger library which essentially create json file conforming to Open API specification based on the annotations. The advantage of this approach will be development team do not have to maintain separate JSON file for documentation, as documentation will be generated from the annotation in the code.

Further Enhancements

1. **Granularity**: Service can be broken down in two services: Account management and Transaction Management Service. This will help to **develop and scale** two services as separate service.
    a. Account Management service: Will be responsible for Create\Update\Delete operation on Accounts.
    b. Transaction Management Service: Will be responsible for Credit\Debit and other operations related to transactions.

2. **External Datastore**: As of now, the data is stored in-memory but to scale the service, there need to be consistent view of the data. To achieve this, we need to store the data in external data store.

3. **Eventual consistency**: Since debit and credit are two operations which need to carried in atomic manner to maintain the consistency of the data. For now, this is achieved by carry out two operations in the try-catch block and reverting transaction if exception occurs from any of the operations. This is alright when the data is in-memory and we are running one service instance only. But when we are running the same operation in scaled up service, above approach will not work, we might to employ other solution like Compensating transaction etc. This will help us to achieve eventual consistency with high availability and latency.

4. **Security**: As of now, the service is not secured. To secure the service, we need to have following:
    a. API gateway: Should be entry point for any service call from external system.
    b. Authentication\Authorization service: Need to have authentication\authorization service which can generate SAML\JWT token based on the request. Further responsibility will be:
        i. Authentication of token
        ii. Authorization of token
        iii. Regeneration of token in case of expiry