Design and code an interest calculation application. It should provide the following endpoints  
   
1. POST processAccountOpening  
This will get account details as input e.g. below. Save this record in your data-store. You could get the same request multiple times so consider idempotency.   
{  
    "bsb": 182182",  
    "identification": 111222333,  
    "openingDate" : "2021-09-13"  
}  
   
2. PUT processAccountEndOfDayBalances  
This will get end-of-day balances for accounts e.g. below. You could get the same request multiple times, so consider idempotency. For each account in your data-store, please calculate daily accrued interest.  
{  
    "balanceDate" : "2021-09-19",  
    [  
        {  
        "bsb": 182182",  
        "identification": 111222333,  
        "balance": 123.34  
        },  
        {  
        "bsb": 182182",  
        "identification": 222000111,  
        "balance": 12.34  
        },  
        {  
        "bsb": 182182",  
        "identification": 222000999,  
        "balance": 0.00  
        }  
    ]  
}  
   
3. PUT calculateMonthlyInterest  
This will be a no-payload endpoint that will sum **up daily interest** amounts per account for the month that’s in request. You can choose to return calculation results if you like.   
   
Stretch: How would you handle account closures ?  
Note 1: Where feasible, we’d like you to write unit tests.  
Note 2: Where feasible, we’d like you to use reactive paradigms.  
Note 3: Provide a supporting doc to explain their choices/unfinished items (readme is fine too).

**Direction Given**

* Deliver working software that meets the requirements.
* Keep the solution as simple as possible whilst meeting the requirements.
* Working unit tests. With dynamic rather than hard coded test data if possible.
* Supporting documentation should explain your vision/solution.
* Manage your time.
* Prioritise and state in documentation what you have prioritised, e.g. parse incoming message into domain object for evaluation would be an important part of the solution to focus on delivering.
* Complete your prioritised features.
* Break down the solution to small iterative parts and make it clear that is the strategy.
* If you realise you will run out of time, stop coding with enough time to document your code (the same way senior developers in a team ensure there is knowledge for others to pick up backlog in their absence).
* All supplied code should work.
* Demonstrate event driven / async models.
* Include thinking in supporting documents.
* Don’t use REST if the strategy calls for async Kafka or other event driven methods.
* Use established practice in coding financial solutions, i.e. BigDecimal, when dealing with payments/interests.
* Uses appropriate coding practice for dealing with financial data - e.g. BigDecimal.
* No lazy assumptions like assuming a month is 30 days.
* No silly defects like replacing running balance with an interest accrued amount.
* Timezones/UTC Offsets,  consider the implications of selecting a timezone.
* No superfluous code / classes
* Clear that a reactive paradigm is used. Only supply code that is used in the solution.
* Use a naming convention that shows organisation/readability/context
* Unit testing has a strategy such as being targeted on risk/business impact, supports CI/CD
* Bonus points - keeping history: e.g. interest accrual history.