

Problem Solution

Modernising the account management system using Microservice architecture is a great initiative aiming to replace the current monolithic, tightly coupled, single application.

Microservices can bring several benefits, like scalability, fault tolerance, and easier maintenance, to this current application.

According to my current knowledge and experience from my previous jobs, below I will propose a solution for the Account Management system using Microservices.

1. Microservices:

I would suggest having two microservices for the Account domain. Each represents a specific business functionality and runs as an independent process. They can communicate with each other via APIs.

a. Account Service.

- Responsible for managing the core functionalities related to the account itself.

b. Transaction Service.

- Responsible for storing and managing transaction data for each account.

Note: Apart from the above, I have also tried to illustrate some theoretical design suggestions for this microservice, and how I started this task by creating a to-do list, and the time I have spent on the same. Which you will find while reading these documents.

1. Domain modeling:

At a high level, we have mainly two main entities in the Account Management system:

- Account and
- Transaction

Attributes	Descriptions
id (Long)	The unique identifier for the account.
accountNumber (String)	The account number associated with the account.
accountType(enum)	Chose Account type while creating a new account i.e., Savings, Fixed, etc.
balance (Double)	The current available balance in the account.

Relationships- Each account can have multiple transactions (**1-to-Many** relationship).

Attributes	Descriptions
id (Long):	The unique identifier for the transaction.
accountNumber (String)	The account number associated with the transaction.
transactionType(enum)	Type of transaction i.e., Deposit, Withdrawal.
amount (double)	Money involved in the transaction.
transactionTime (LocalDateTime)	The timestamp of when the transaction occurred.

Relationships- Each transaction is associated with a single account (**Many-to-1** relationship).

2. Use cases:

The reason for defining the use cases is to capture the functional requirements of the Account Management system by describing the interactions between external entities, i.e., customers, and what they will achieve after interacting with the Account Management system.

Actor:

The Customer is an external entity that interacts with the Account Management system.

Use Cases	Descriptions
1. Create Account	The customer can create a new savings account through the Account Service.
2. Deposit Money	The customer can deposit money into their account using the Account service.
3. Withdraw Money	The Customer can withdraw money from their account using the Account Service.
4. View Balance	The Customer can check the available balance in their account using the Account Service.
5. View Account	The Customer can view their account details, including the balance and the last 10 transactions using the Account Service.

3. API Design:

I have used RESTful API design principles to define the endpoints.

1. API's Endpoint design for Account Service

Rest API's	Description
Use case 1: Create Account Endpoint: 'POST /accounts' Request Body: (AccountDTO) { "accountType": "saving", "balance": 0.0 } Response Body: (AccountDTO) { "id": 5001, "accountNumber": "5234567859", "accountType": "saving", "balance": 0.0 }	This endpoint will be responsible for creating a savings account for a customer. It will receive necessary details like customer information, account type, etc. It will generate a unique account number and store the account information in the database.
Use case 2: Deposit Money Endpoint: 'POST /accounts/{accountNumber}/deposit' Request Body: (AccountDTO) { "accountNumber": "5234567859", "amount": 100 } Response Body: (AccountDTO) { "id": 5001, "accountNumber": "5234567859", "balance": 100.0 }	This endpoint allows customers to deposit money into their savings account. It will receive the account number and the amount to deposit. The service will update the account's balance accordingly.
Use case 3: Withdraw Money Endpoint: 'POST /accounts/{accountNumber}/withdraw' Request Body: (AccountDTO) { "accountNumber": "5234567859", "amount": 50 } Response Body: (AccountDTO) {	This endpoint allows customers to withdraw money from their savings account. It will receive the account number and the amount to withdraw. The service will check if the withdrawal amount is valid based on the account balance. If the withdrawal is valid, it will update the account balance.

<pre>"id": 5001, "accountNumber": "5234567859", "balance": 50.0 }</pre>	
Use case 4: View Balance Endpoint: 'GET /accounts/{accountNumber}/balance' Request Body: (AccountDTO) <pre>{ "accountNumber": "5234567859" }</pre> Response Body: (AccountDTO) <pre>{ "amount": 50.0 }</pre>	<p>This endpoint allows customers to retrieve the available balance in their savings account.</p> <p>It will receive the account number and return the account balance.</p>
Use case 5: View Account Endpoint: 'GET /accounts/{accountNumber}/ transactions' Request Body: (None required) <pre>{ "accountNumber": "5234567859" }</pre> Response Body: (List of TransactionDTOs) <pre>[{ "id": 5001, "accountNumber": "5234567859", "amount": 50.0, "transactionTime": "2023-07-27T08:19:15" }, { "id": 5002, "accountNumber": "5234567859", "amount": 25.0, "transactionTime": "2023-07-29T09:11:13" } up to 8 more...if more record exists]</pre>	<p>This endpoint allows customers to view their last 10 transactions.</p> <p>It will receive the account number and fetch the transaction history from the database.</p> <p>The service will then return the list of transactions sorted by the timestamp.</p>

2. API's Endpoint design for Transaction Service

Rest API's	Description
Endpoint: 'POST / transactions' Request Body: (Transaction) Response Body: (Transaction) <pre>{ "id": 5001, "accountNumber": "5234567859", "balance": 0.0 }</pre>	<p>This endpoint will be called internally by the Account Service whenever a deposit or withdrawal occurs.</p> <p>It will record the transaction details, such as amount, type (deposit/withdrawal), and timestamp.</p>

3. Error Handling:

For invalid input and business validations, appropriate HTTP status codes would be returned along with meaningful error messages in the response body. For example, when an invalid account number is provided, the response could be:

```
{
  "message": "Invalid input or Invalid account number provided."
}
```

4. API Documentation:

Swagger can be used to document REST APIs. By adding Swagger annotations to the controller methods, an interactive API documentation page would be generated. This documentation would provide details about each endpoint, request parameters, response types, and possible error responses.

5. Realization

Since the Account service and Transaction service are independent, they will communicate through RESTful endpoints. I would also suggest implementing proper authentication and authorization mechanisms to secure the services and the customer data.

Since the Customer domain is managed by other systems within the bank, as far as I can understand, we can basically interact with those systems through appropriate APIs to fetch customer details when needed. This way, we can maintain a clear separation of concerns and leverage the existing functionality without duplicating efforts.

Finally, with the earlier mentioned use cases, domain modelling, and API design, we can have an understanding of the data entities, their relationships, and the endpoints needed to implement the Account Management system using the Microservice architecture.

2. API Gateway:

To provide unified interface to the clients, I would suggest to have an API gateway that sits in front of above mentioned two microservices which will handle incoming requests from clients. Basically, it routes the request to the appropriate microservice.

3. Database:

Good to have its own database for storing its specific data. For simplicity, we can use separate databases for the Account and Transaction services.

Sometime it might not be possible to have separate database at the initial phase of the project development. It depends on the complexity of data. As I mentioned during our talk. In my previous job where I had a very tough time to separate databases for each microservices due to so many relationships among tables, etc.

4. Event

To ensure loose coupling between microservices, good to implement an event-driven architecture using a message broker (e.g., Apache Kafka or RabbitMQ). For example, when a new transaction is made, the Transaction Service can publish an event, and the Account Service can subscribe to it to update the account balance. It's not that we should have it during a start but nice to keep it open for the future improvement.

5. Authentication and Authorization

To ensure that only authorized users can access the services.

6. Monitoring and logging

Good to Implement logging and monitoring mechanisms to track the performance and health of each microservices. Maybe good to use Grafana for visualization and Prometheus as a data source.

7. Containerization

Recommended to use technologies like Docker for (packing code + dependencies) and Kubernetes (to manage containers) for easy deployment and scalability.

8. Final thought, to-do list, and Time spent

To-Do:

- Understand & analyze given task requirements
- Describes the entities involved (Domain modeling)
- Define functional requirements (maybe non-functional requirements later for System design)
- Identify number of microservices are needed
- Define schema
- Code design
- Design Architecture in terms of layer
- Implementation
- Test
- Document

Time spent:

Total time has been spent approx. 5hrs, for the documentation 2-3 hrs., implementation 2-3 hrs., it was recommended to spend max. 3 hrs., but after starting the task I didn't realize the hours.

Time spends day and time wise:

Saturday: morning 6:00 to 9:00 (including a coffee break) --- 3 hrs.

Sunday: morning 6:00 to 8:00 (No break) --- 2 hrs.

Final thought

I only designed and wrote code at a high level. I did not manage to spend much time on fine-grained details. Even for the implementation, I have written very high-level code to reflect my earlier discussions.

9. Implementation screenshot

The screenshot was not mandatory to attach here since I have already attached the source code along with the email. I just attached them for reference for both services.

1. Account Service Implementation screenshots

Entity: Account

```
14 @Entity
15 @Table(name = "account")
16 public class Account {
17
18     @Id
19     @GeneratedValue(strategy = GenerationType.AUTO)
20     private Long id;
21
22     @Column(name = "account_type")
23     @Enumerated(EnumType.STRING)
24     private AccountType accountType;
25
26     @Column(name = "account_number", unique = true, nullable = false)
27     private String accountNumber;
28
29     private Double balance;
30 }
```

Service: Account Service and Their Implementation

```
8 public interface AccountService {
9
10     /**
11      * Creates a new savings account for the customer.
12      *
13      * @param accountDTO for which account needs to be created
14      * @return The created AccountDTO.
15      */
16     AccountDTO createAccount(AccountDTO accountDTO);
17
18     /**
19      * Deposits money to the specified account.
20      *
21      * @param accountNumber for which money needs to be deposited.
22      * @param amount (money) to be deposited.
23      * @return updated AccountDTO.
24      */
25     AccountDTO depositMoney(String accountNumber, double amount);
26
27     /**
28      * Withdraws money from the provided account.
29      *
30      * @param accountNumber from which money needs to be withdrawn.
31      * @param amount (money) to be withdrawn.
32      * @return updated AccountDTO.
33      */
34     AccountDTO withdrawMoney(String accountNumber, double amount);
35
36     /**
37      * Retrieves the available balance for the provided account.
38      *
39      * @param accountNumber for which available balance needs to be fetched.
40      * @return available balance as a double value.
41      */
42     double getAvailableBalance(String accountNumber);
43
44     /**
45      * Retrieves the last 10 transactions for the provided account.
46      *
47      * @param accountNumber for which transactions need to be fetched.
48      * @return A list of TransactionDTO representing the last 10 transactions.
49      */
50     List<TransactionDTO> getLast10Transactions(String accountNumber);
51 }
```

```

22 @Service
23 public class AccountServiceImpl implements AccountService {
24
25     @Autowired
26     private AccountRepository accountRepository;
27
28     // temporary storage, replace with accountRepository for real db in future
29     private Map<String, Account> accounts = new HashMap<>();
30     private Map<String, List<Transaction>> transactions = new HashMap<>();
31
32     @Override
33     @Transactional
34     public AccountDTO createAccount(AccountDTO dto) {
35
36         try {
37             Account account = new Account();
38
39             String accountNumber = AccountUtils.generateRandomAccountNumber();
40             double initialBalance = 0.0;
41             account.setId(100 + (long) (Math.random() * 899));
42             account.setAccountNumber(accountNumber);
43             account.setBalance(initialBalance);
44             AccountType accountType = AccountType.valueOf(dto.getAccountType().toUpperCase());
45             account.setAccountType(accountType);
46
47             // Validate allowed values for accountType: SAVINGS, FIXED, CURRENT
48             switch (accountType) {
49                 case SAVINGS, FIXED, CURRENT -> accounts.put(accountNumber, account);
50
51                 default -> throw new AccountServiceException("Invalid account type: " + dto.getAccountType());
52             }
53
54             return AccountMapper.mapAccountToDTO(account);
55         } catch (Exception e) {
56             throw new AccountServiceException("Invalid input: " + e.getMessage());
57         }
58     }
59 }
60

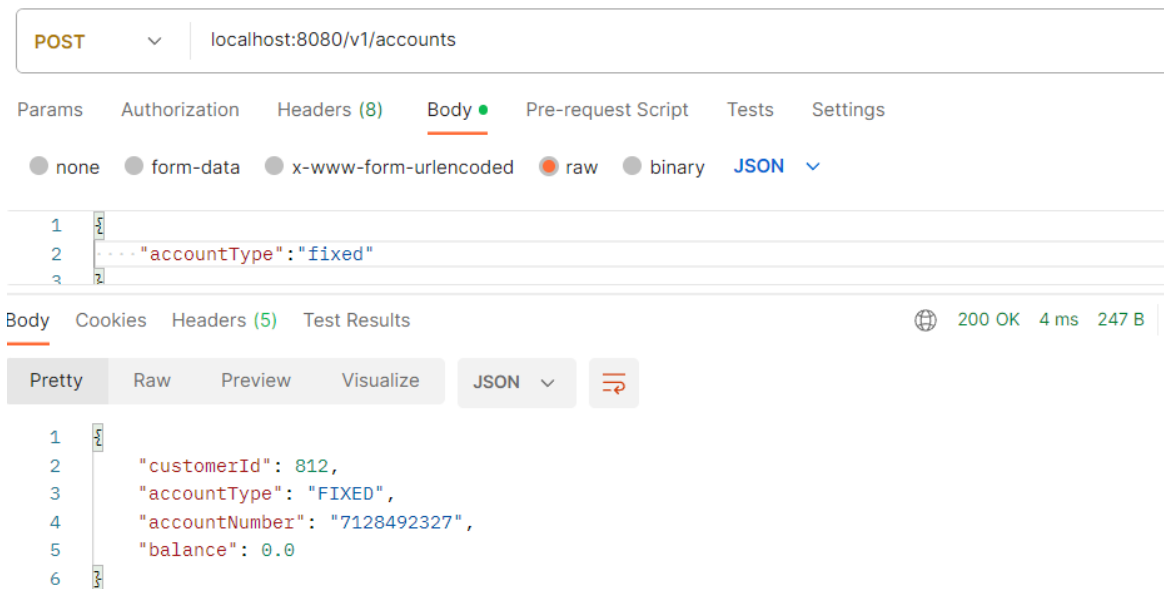
```

```

20 @RestController
21 @RequestMapping("/v1/accounts")
22 public class AccountResource {
23
24     @Autowired
25     private AccountService accountService;
26
27     @PostMapping
28     public ResponseEntity<AccountDTO> createAccount(@Valid @RequestBody AccountDTO accountDto) {
29         AccountDTO createdAccount = accountService.createAccount(accountDto);
30         return ResponseEntity.ok(createdAccount);
31     }
32
33     @PostMapping("/{accountNumber}/deposit")
34     public ResponseEntity<AccountDTO> depositMoney(@PathVariable String accountNumber,
35         @RequestBody TransactionDTO transactionDTO) {
36         AccountDTO updatedAccount = accountService.depositMoney(accountNumber, transactionDTO.getAmount());
37         return ResponseEntity.ok(updatedAccount);
38     }
39
40     @PostMapping("/{accountNumber}/withdraw")
41     public ResponseEntity<AccountDTO> withdrawMoney(@PathVariable String accountNumber,
42         @RequestBody TransactionDTO transactionDTO) {
43         AccountDTO updatedAccount = accountService.withdrawMoney(accountNumber, transactionDTO.getAmount());
44         return ResponseEntity.ok(updatedAccount);
45     }
46
47     @GetMapping("/{accountNumber}/balance")
48     public ResponseEntity<Double> getAvailableBalance(@PathVariable String accountNumber) {
49         double balance = accountService.getAvailableBalance(accountNumber);
50         return ResponseEntity.ok(balance);
51     }
52
53     @GetMapping("/{accountNumber}/transactions")
54     public ResponseEntity<List<TransactionDTO>> getLast10Transactions(@PathVariable String accountNumber) {
55         List<TransactionDTO> transactions = accountService.getLast10Transactions(accountNumber);
56         return ResponseEntity.ok(transactions);
57     }
58 }

```


Test via the Postman client:



Test by code:

```
13
14 @SpringBootTest
15 public class AccountResourceTest {
16
17     @Autowired
18     private AccountResource accountResource;
19
20     @Test
21     public void testCreateAccount() {
22         AccountDTO dto = new AccountDTO();
23
24         // Test with valid account type: SAVINGS, expected to pass
25         dto.setAccountType("savings");
26         ResponseEntity<AccountDTO> accountDTO = accountResource.createAccount(dto);
27
28         assertNotNull(accountDTO);
29         assertEquals(10, accountDTO.getBody().getAccountNumber().length());
30         assertEquals(0.0, accountDTO.getBody().getBalance());
31     }
32 }
```

Transaction Service Implementation Screenshots

```
16 @Entity
17 @Table(name = "transaction")
18 public class Transaction {
19
20     @Id
21     @GeneratedValue(strategy = GenerationType.AUTO)
22     private Long id;
23
24     private String accountNumber;
25
26     @Positive(message = "Amount must be positive")
27     private Double amount;
28
29     @Enumerated(EnumType.STRING)
30     private TransactionType type;
31
32     private LocalDateTime transactionTime;
33
34 }
```



```

7 public interface TransactionService {
8     /**
9      * Creates a new transaction for the provided account.
10     *
11     * @param transactionDTO contains transaction details.
12     * @return The created TransactionDTO.
13     */
14     TransactionDTO createTransaction(TransactionDTO transactionDTO);
15
16     /**
17     * Retrieves the last 10 transactions for the specified account.
18     *
19     * @param accountNumber for which transactions need to be fetched.
20     * @return A list of TransactionDTO representing the last 10 transactions.
21     */
22     List<TransactionDTO> getLast10Transactions(String accountNumber);
23 }
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111 @Service
112 public class TransactionServiceImpl implements TransactionService {
113
114     @Autowired
115     private TransactionRepository transactionRepository;
116
117     @Override
118     public TransactionDTO createTransaction(TransactionDTO transactionDTO) {
119         // Implement the logic to create a new transaction and save it to the repository
120         // Map the TransactionDTO to the entity and vice versa
121         // Return the created TransactionDTO
122         return transactionDTO;
123     }
124
125     @Override
126     public List<TransactionDTO> getLast10Transactions(String accountNumber) {
127         // Implement the logic to retrieve the last 10 transactions for the provided
128         // Map the entities to TransactionDTOs
129         // Return the list of TransactionDTOs
130         return null;
131     }
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```