

Project Report

Group 2e:

梁家铭(151250091), 李一然(151250087), 倪辰皓(141250096), 曹鸿荣(151250006),
常德隆(151250011), 陈进(151250013), 陈淼(151250015), 陈锐(151250016)

1.	Project Overview	3
1.1	Major capabilities.....	3
1.2	Operational scenarios	3
2.	Architecture Options.....	4
2.1	Microservices architecture pattern	4
2.2	MVC architecture pattern	5
3.	ADD Procedures for Microservices Pattern.....	6
3.1	Iteration 1.....	6
3.2	Iteration 2.....	10
3.3	Iteration 3.....	11
3.4	Iteration 4.....	13
3.5	Iteration 5.....	16
3.6	Iteration 6.....	18
3.7	Final architectural views.....	20
3.7.1	Module view.....	20
3.7.2	Component-Connector view	20
4.	ADD Procedures for MVC Pattern	21
4.1	Iteration 1.....	21
4.2	Iteration 2.....	21
4.3	Iteration 3.....	23
4.4	Iteration 4.....	25
4.5	Iteration 5.....	26
4.6	Final architectural views.....	27
4.6.1	Module view.....	27
4.6.2	Component-Connector view	27
5.	Architecture Patterns Comparison	28
5.1	Overview	28
5.2	Performance.....	28
5.3	Availability	28
5.4	Security.....	28
5.5	Cost.....	29
5.6	Conclusion	29
6.	UML Class Diagram and Mapping	29
6.1	Class Diagram	29
6.2	Class Mapping	30
7.	ATAM Procedures for Microservices Pattern.....	30

7.1	Quality Attribute Utility Tree	30
7.2	ATAM Analysis	31
7.3	Sensitivity points	34
7.4	Tradeoff points	35
7.5	Risk	36
7.6	Non-risk	36
8.	Summary	37
8.1	Challenges and experience.....	37
8.2	Individual's contributions.....	37

1. Project Overview

1.1 Major capabilities

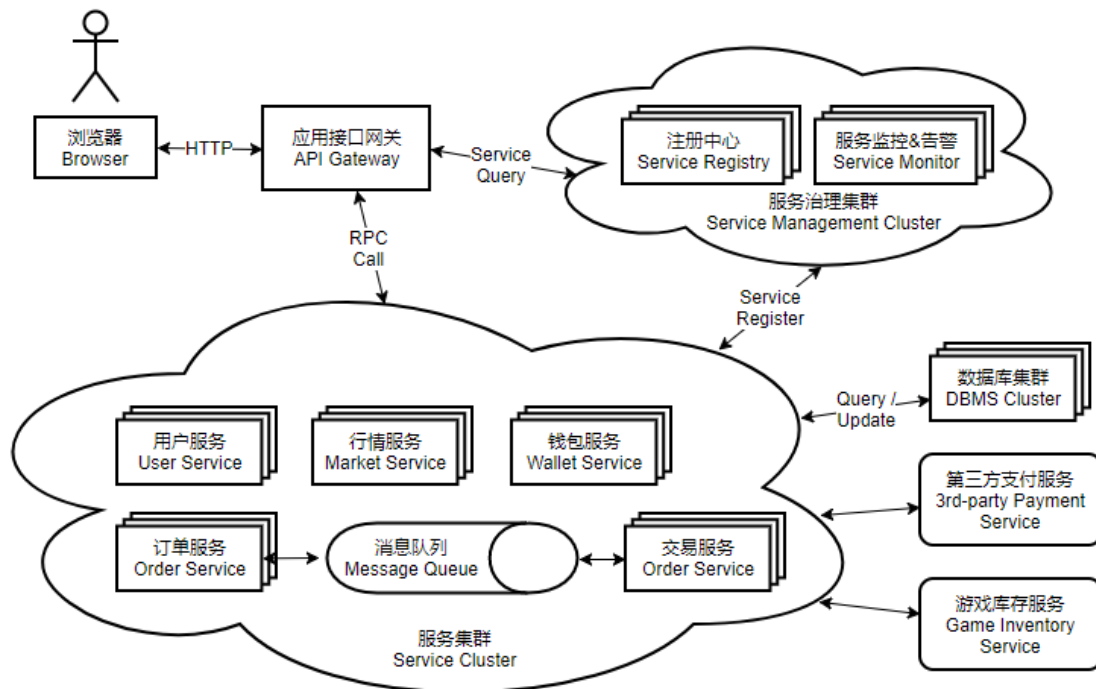
- 1) Items browsing: Users can browse the list of items for sale and sort them by price or trading volume.
- 2) Market quotations: Users can view quotations of a certain item, including the numbers of buy and sell orders, lowest ask price, highest bid price, recent transaction prices, trends of price and trading volume, etc.
- 3) Buy order: Users can set the bid price for a certain item and create a buy order. Trades are made automatically when a sell order with an ask price lower than or equal to the bid price occurs.
- 4) Sell order: Users can set the ask price for a certain item in his inventory and create a sell order. Trades are made automatically when a buy order with a bid price higher than or equal to the ask price occurs.
- 5) Order management: Users can view all his buy and sell orders, and cancel orders that are not yet completed.
- 6) Wallet: Each user has a personal wallet. Users can view income and expense records, add funds to account or withdraw from account (via third-party payment platforms).
- 7) Inventory: Users can check his own inventory of in-game items (via game operators).

1.2 Operational scenarios

Via in-game item trading system, users can sell their own items at an arbitrary ask price or quote certain items at an arbitrary bid price by issuing an order. The system automatically matches the orders so that trades are made when an ask price is fulfilled by a certain bid price.

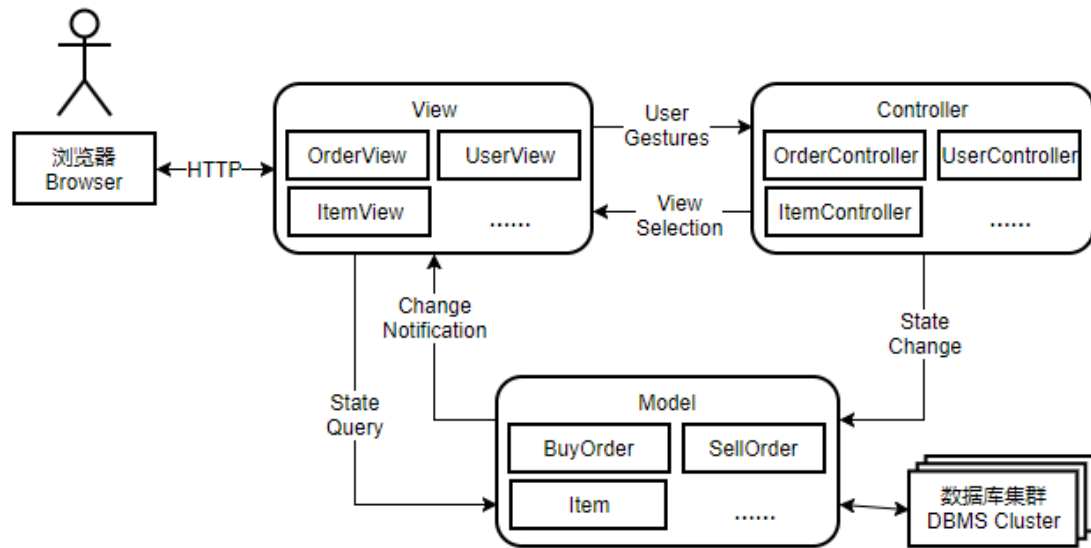
2. Architecture Options

2.1 Microservices architecture pattern



Distributed microservices architecture, which is widely applied nowadays, is adopted in this pattern. Here the service management cluster is responsible for the registration, naming and monitoring, while the highly decoupled service cluster discomposes the business functions into multiple services each of which only assumes its own duties, thereby improving modifiability and extensibility. API Gateway handles all external interfaces, accepting requests from users' browsers and re-assembling the responses accordingly. High-efficiency RPC protocols are used to implement remote calls between API Gateway and services. Moreover, service cluster supports software load-balancing and hot swapping features, which enables the system to elevate performance and availability and to sustain high concurrency situations.

2.2 MVC architecture pattern



MVC design pattern helps minimize the coupling between modules, where model, view and controller are three independent components so that one of them can be modified without affecting any other one. Reusability is enhanced since different views can access a single model component simultaneously. Meanwhile, MVC architecture pattern enables quick development and quick deployment with high maintainability and extensibility, which is beneficial to software engineering management and life cycle cost optimization.

3. ADD Procedures for Microservices Pattern

3.1 Iteration 1

3.1.1 Requirements information

3.1.1.1 Functional requirements

See also 1.1 Major capabilities.

3.1.1.2 Scenarios

Scenario 1: >1000 users trading at the same time [Reliability, Performance]

Element	Possible values
Source	User
Stimulus	>1000 users trading at the same time
Artifact	Load balancing module, business module, database system
Environment	System functioning properly
Response	System can execute purchase operations by users normally Server delivers large-scale requests to computational nodes Data stored in database are modified normally
Response measures	>99% of user requests are properly handled >85% of user requests are handled within 1s

Scenario 2: Normal trade operation by user [Usability, Performance]

Element	Possible values
Source	User
Stimulus	User performing normal trade operation
Artifact	System
Environment	System functioning properly
Response	User operation is guided correctly Operation is simple and easy to learn
Response measures	Finishing simple, medium and complicated tasks take no longer than 20s, 1min and 2min respectively >99% of users can use system correctly >90% of users can get familiar with system operations within 1h

Scenario 3: Trade operation by unauthorized user [Security]

Element	Possible values
Source	Unauthorized user
Stimulus	Making a payment (or other security-critical operations)
Artifact	System security module
Environment	System functioning properly, user not authorized
Response	User request rejected by system System generates log

Response measures	System rejects illegal user request within 2s System generates log instantly after rejecting request
--------------------------	---

Scenario 4: Invalid user input [Robustness]

Element	Possible values
Source	User
Stimulus	Invalid input from user
Artifact	Input validation module
Environment	System functioning properly
Response	System displays error information System provides possible solutions
Response measures	System recognizes invalid input and responses correctly within 2s

Scenario 5: Adding new feature to system [Extensibility, Maintainability]

Element	Possible values
Source	Developer
Stimulus	New feature to be added to system
Artifact	Business computing module
Environment	System already released and functioning properly
Response	System releases new feature Release process should be simple and easy
Response measures	Over 99% of users remain unaffected during system release Release of new feature should be accomplished within 1h

Scenario 6: Changing current system function [Modifiability]

Element	Possible values
Source	Developer
Stimulus	Developer want to modify existent feature
Artifact	System
Environment	During development
Response	Implement necessary changes Test modified code Deploy modified system modules
Response measures	Changing one system function costs under 2 man-months Modification costs over 80% less than rebuilding system

Scenario 7: Increasing/decreasing computing power [Scalability]

Element	Possible values
Source	Developer
Stimulus	Computing power needs to be increased or decreased for certain business
Artifact	Business computing module
Environment	System already released and functioning properly

Response	Computing power is increased or decreased correspondingly
Response measures	Change of computing power is accomplished within 1h Over 99% of users remain unaffected during computing power increase/decrease

Scenario 8: Migrating system to another environment [Portability]

Element	Possible values
Source	Developer, maintainer
Stimulus	Migrating system to another environment
Artifact	System
Environment	During development, maintenance or configuration
Response	Finish necessary modifications for system migration Test modified modules Deploy system to new environment
Response measures	System migration costs under 2 man-months Migration costs over 80% less than rebuilding system

Scenario 9: Server down [Availability]

Element	Possible values
Source	Maintainer
Stimulus	Server down for service
Artifact	Service cluster
Environment	Server functioning properly, under development or maintenance
Response	Identify server error Restart server
Response measures	Server starts working normally within 1 minute

Scenario 10: Failed service discovery [Reliability, Availability]

Element	Possible values
Source	Maintainer
Stimulus	Certain service cannot be discovered
Artifact	Server, communication module, error handling module
Environment	System functioning properly or under test
Response	Check for connection failure, and restore connection if any Check for server error, and resolve error if any
Response measures	Service discovery becomes available within 6h Error checking and restoration cost under 2 man-months

Scenario 11: Failed service registration [Reliability, Availability]

Element	Possible values
Source	Developer
Stimulus	Service cannot be registered
Artifact	Communication module, service cluster module, error handling

	module
Environment	System functioning properly or under test
Response	Analyze cause of issue Solve problems and restore service registration
Response measures	Service registration becomes available within 6h Error checking and restoration cost under 2 man-months

Scenario 12: Database crash [Availability, Safety]

Element	Possible values
Source	Maintainer
Stimulus	Database crash
Artifact	Database system, error handling module
Environment	System functioning properly or under test
Response	Identify cause of crash Restore database system
Response measures	Database becomes available within 6h Error checking and restoration cost under 2 man-months

Scenario 13: Network outage [Reliability, Robustness]

Element	Possible values
Source	User, developer
Stimulus	Network connection failure
Artifact	Communication module, error handling module
Environment	System functioning properly or under test
Response	Check for network connection problem Check for communication module malfunction And restore connection
Response measures	Service registration becomes available within 6h Error checking and restoration cost under 2 man-months

Scenario 14: Checking status information of service cluster [Reliability, Performance]

Element	Possible values
Source	Maintainer
Stimulus	Checking status information of service cluster
Artifact	Service cluster status log
Environment	System functioning properly
Response	System generates service cluster status log
Response measures	System can generate service cluster status log of any time period Accuracy of service cluster status log is at least 99.9%

3.1.2 Decomposed system components

Iteration 1 is for the overall architecture design. No system components are decomposed at this

iteration.

3.1.3 Element view diagram

See also 2.1 Microservices architecture pattern.

3.2 Iteration 2

3.2.1 Requirements information

Requirements information is identical to Iteration 1.

3.2.2 Decomposed system components

The Service Registry component is decomposed at this iteration. Service Registry is responsible for service registering and discovering.

3.2.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 5: Adding new feature to system	M	H
Scenario 7: Increasing/decreasing computing power	M	M
Scenario 10: Failed service discovery	M	M
Scenario 11: Failed service registration	M	M

3.2.4 Designing for ASRs

3.2.4.1 Design concerns

Quality Attribute	Design Concern	Subordinate Concerns
Availability, Performance	Service registering and discovering	Synchronization between nodes
Extensibility, Scalability	Service addition and deletion	Construction of the registry

3.2.4.2 Alternative patterns for concerns

1) Synchronization between nodes

#	Pattern	Availability	Performance	Cost
1	Use only one node for Service Registry	L	L	L
2	Consistency & Availability pattern	H	M	M
3	Availability & Partition tolerance pattern	H	H	M
4	Consistency & Partition tolerance pattern	L	M	M

In theoretical computer science, the CAP theorem states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: Consistency, Availability and Partition tolerance.

Pattern #3 is selected. It provides both high availability and high performance at the same cost. ZooKeeper and its ZAB protocol, which is widely-used to build distributed data stores, can be used to achieve the goal of this pattern.

2) Construction of the registry

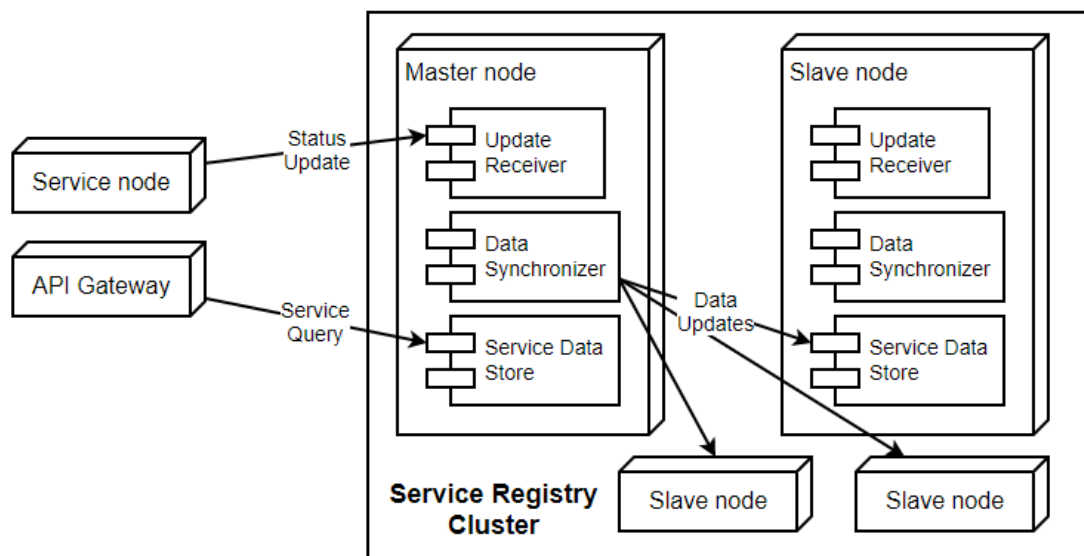
#	Pattern	Availability	Performance	Cost
1	Services push updates to the registry	M	H	L
2	Service Registry pulls updates from services	H	L	M

Pattern #1 is selected. Though pattern #2 can achieve higher availability, it consumes more resources (network, etc.) and cost more while developing. Pattern #2 can be selected in the Service Monitoring cluster, instead of the Service Registry.

3.2.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
Synchronization between nodes	Availability & Partition tolerance pattern	Scenario 7, Scenario 9, Scenario 10
Construction of the registry	Services push updates to the registry	Scenario 5, Scenario 11

3.2.5 Element view diagram



3.2.6 Evaluation

No conflicts were found during the evaluation of this iteration.

3.3 Iteration 3

3.3.1 Requirements information

Requirements information is identical to Iteration 1.

3.3.2 Decomposed system components

The Service Monitor component is decomposed at this iteration. Service Monitor is responsible for monitoring service availability and apply failover mechanism when service unavailability is

detected. It ensures the performance, availability and reliability of services, which is significant to the operation of the whole system.

3.3.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 1: >1000 users trading at the same time	H	H
Scenario 9: Server down	H	M
Scenario 13: Network outage	M	M
Scenario 14: Checking status information of service cluster	M	M

3.3.4 Designing for ASRs

3.3.4.1 Design concerns

Quality Attribute	Design Concern	Subordinate Concerns
Availability	Service availability monitoring	Service status monitoring
Availability	Service unavailability handling	Unavailability handling method
Performance	Service performance monitoring	Response time measure, Service invocation counting

3.3.4.2 Alternative patterns for concerns

1) Service status monitoring

#	Pattern	Availability	Service Cost	Monitor Cost
1	Heartbeat	H	H	M
2	Ping/echo	H	M	H

Pattern #1 is selected. It lightens the stress of services, which is vital to the whole system.

2) Unavailability handling method

#	Pattern	Availability	Success Rate	Cost
1	Report and try automatic reboot	H	M	L
2	Record and wait for manual reboot	L	H	M

Pattern #1 is selected. Automatic reboot pattern reduces manpower consumption and saves cost of the system. Manual reboot is still considered, but is only needed when automatic reboot is not working.

3) Response time measure

#	Pattern	Accuracy	Performance
1	Measure response time of each request	H	M
2	Measure response time of random requests	M	H

Pattern #2 is selected. Measuring each request consumes much more system resources and may affect normal operations. Measuring random requests consumes less resources, and its accuracy depends on the algorithm to select requests randomly.

4) Service invocation counting

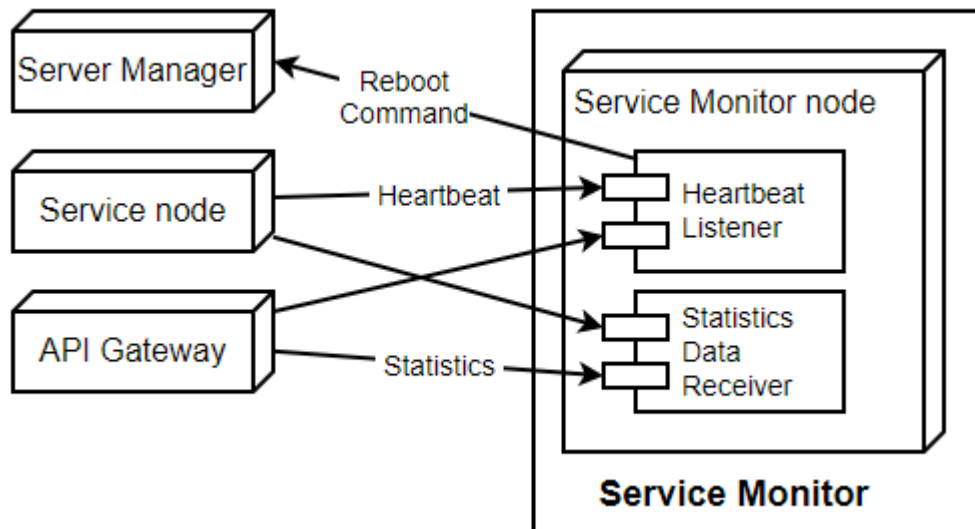
#	Pattern	Accuracy	Performance
1	Count on each invocation	H	M
2	Count on a regular interval	M	H
3	Count on a regular amount of invocations	M	H

Pattern #2 is selected. Counting on each invocation consumes lots of system resources, while the rest patterns cost less. Counting on a regular amount of invocations may lead to accuracy decrement if the service is invoked irregularly. Counting on a regular interval ensures the data collected regularly.

3.3.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
Service status monitoring	Heartbeat	Scenario 1, Scenario 9, Scenario 13
Unavailability handling method	Report and try automatic reboot	Scenario 1, Scenario 9, Scenario 13
Response time measure	Measure response time of random requests	Scenario 14
Service invocation counting	Count on a regular interval	Scenario 14

3.3.5 Element view diagram



3.3.6 Evaluation

No conflicts were found during the evaluation of this iteration.

3.4 Iteration 4

3.4.1 Requirements information

Requirements information is identical to Iteration 1.

3.4.2 Decomposed system components

The API Gateway component is decomposed at this iteration. API Gateway receives HTTP requests from users' browsers, then forwards them to the Service Cluster by Remote Procedure Calls (RPCs). It is also a vital part of the whole system.

3.4.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 1: >1000 users trading at the same time	H	H
Scenario 2: Normal trade operation by user	H	M
Scenario 9: Server down	M	M
Scenario 10: Failed service discovery	M	M

3.4.4 Designing for ASRs

3.4.4.1 Design concerns

Quality Attribute	Design Concern	Subordinate Concerns
Availability	Failover mechanism	Service unavailability discovery, Service unavailability handling
Availability	Service discovery	Maintain connection with Service Registry
Performance	Service load balancing	Load balancing pattern
Performance	Caching	Caching method

3.4.4.2 Alternative patterns for concerns

1) Service unavailability discovery

#	Pattern	Availability	Response Time
1	Service unavailable if an amount of requests failed	H	M
2	Service unavailable if no success requests in an amount of time	H	L

Pattern #1 is selected. In a real environment, the frequency of requests is hard to predict. Therefore, Pattern #2 can easily cause false alarms, though its low response time.

2) Service unavailability handling

#	Pattern	Availability	Response Time
1	Return cached result or error message immediately (fail-fast)	H	L
2	Retry until service is available again (fail-back)	M	H
3	Return cached result or error message after an amount of retries (fail-over)	H	M

Pattern #3 is selected. In this pattern, both retrying and response time is considered. Limiting the amount of retries makes the response time more predictable.

3) Maintain connection with Service Registry

#	Pattern	Availability	Cost
1	Use persistence connection and heartbeat	H	H
2	Ping Service Registry when requests arrive	M	M

Pattern #1 is selected. Though its high cost, it effectively ensures high availability.

4) Load balancing pattern

#	Pattern	Performance	Cost
1	Weighted round robin	M	L
2	Least response time	H	M
3	Consistent hashing	M	L
4	Random	L	L

Pattern #2 is selected. In a trading system, response time is significant. However, random method can be used in other systems. "Scientific research shows that random often creates miracles." said Tongwei Ren, associate professor at Software Institute, Nanjing University.

5) Caching method

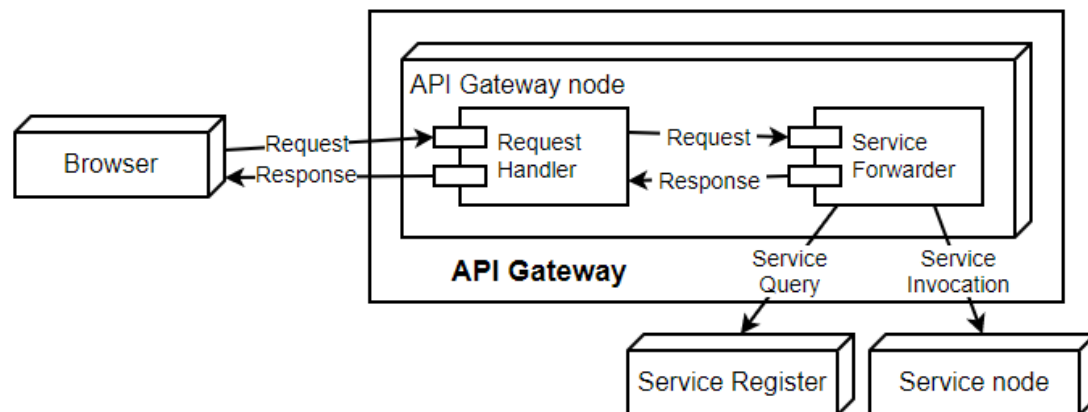
#	Pattern	Availability	Performance	Cost
1	In-memory caching	M	H	L
2	Dedicated caching server (Redis, Memcached, ...)	H	M	M

Pattern #2 is selected. In-memory caching is not robust and often causes data loss, thus a dedicated caching server is needed for high availability.

3.4.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
Service unavailability discovery	Service unavailable if an amount of requests failed	Scenario 1, Scenario 9
Service unavailability handling	Return cached result or error message after an amount of retries (fail-over)	Scenario 1, Scenario 9
Maintain connection with Service Registry	Use persistence connection and heartbeat	Scenario 10
Load balancing pattern	Least response time	Scenario 1
Caching method	Dedicated caching server (Redis, Memcached, ...)	Scenario 1, Scenario 2

3.4.5 Element view diagram



3.4.6 Evaluation

No conflicts were found during the evaluation of this iteration.

3.5 Iteration 5

3.5.1 Requirements information

Requirements information is identical to Iteration 1.

3.5.2 Decomposed system components

The Service Cluster component is decomposed at this iteration. Service Cluster handles service invocations by Remote Procedure Calls (RPCs).

3.5.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 1: >1000 users trading at the same time	H	H
Scenario 2: Normal trade operation by user	H	M
Scenario 8: Migrating system to another environment	L	H

3.5.4 Designing for ASRs

3.5.4.1 Design concerns

Quality Attribute	Design Concern	Subordinate Concerns
Performance, Portability	Data transferring protocol	Network protocol, Data serialization method, Data compression method

3.5.4.2 Alternative patterns for concerns

1) Network protocol

#	Pattern	Performance	Portability	Difficulty
---	---------	-------------	-------------	------------

1	Use HTTP protocol for data transferring	H	H	L
2	Use custom TCP protocol for data transferring	H	L	H

Pattern #1 is selected. HTTP is a widely-used and mature protocol for data transferring. It can handle both short connections and persistence connections. Custom TCP protocol cost time for developing, and it needs to be ported to new environment when the system is being migrated to another environment.

2) Data serialization method

#	Pattern	Performance	Portability	Difficulty
1	Use JSON serialization	M	H	M
2	Use cross-language binary serialization (Hession, Protostuff, ...)	H	M	L
3	Use Java-only binary serialization (JVM, Kryo, ...)	H	L	L

Pattern #2 is selected. Cross-language binary serialization methods achieve balance of performance, portability and difficulty. It is fast, easy to use, and needs less time when migrating to another environment.

3) Data compression method

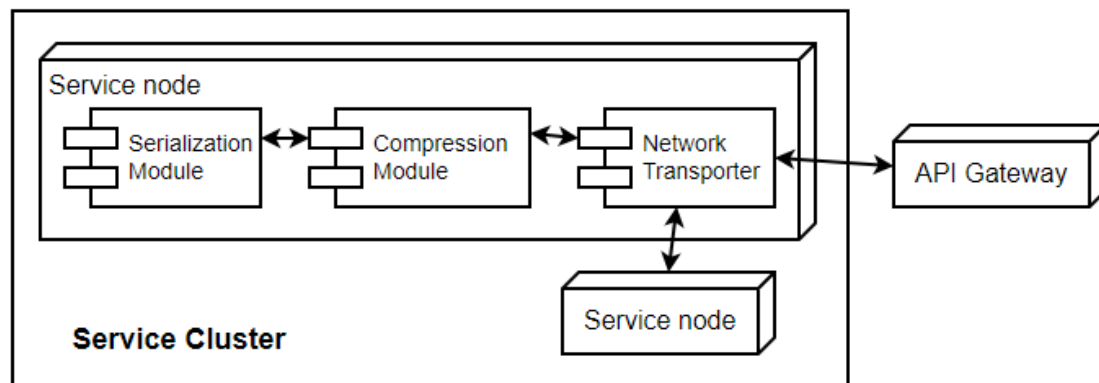
#	Pattern	Performance	Portability	Difficulty
1	No data compression	M	H	L
2	Compress all data	M	M	L
3	Compress data larger than specific size	H	M	M

Pattern #3 is selected. If no data is compressed, large packets needs more time to be transferred on the network. If all data is compressed, more system resources are consumed for small packets. In a real environment, most of RPC requests are small packets, thus pattern #3 achieves high performance for both small and large packets.

3.5.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
Network protocol	Use HTTP protocol for data transferring	Scenario 1, Scenario 2, Scenario 8
Data serialization method	Use cross-language binary serialization (Hession, Protostuff, ...)	Scenario 1, Scenario 2, Scenario 8
Data compression method	Compress data larger than specific size	Scenario 1, Scenario 2, Scenario 8

3.5.5 Element view diagram



3.5.6 Evaluation

No conflicts were found during the evaluation of this iteration.

3.6 Iteration 6

3.6.1 Requirements information

Requirements information is identical to Iteration 1.

3.6.2 Decomposed system components

The Security component is decomposed at this iteration. This component is responsible for user permission verification and input data filtering.

3.6.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 3: Trade operation by unauthorized user	H	M
Scenario 4: Invalid user input	H	M
Scenario 12: Database crash	H	H

3.6.4 Designing for ASRs

3.6.4.1 Design concerns

Quality Attribute	Design Concern	Subordinate Concerns
Security	Attack prevention	User authentication, Input data validation
Security	Network transport security	Encryption method
Safety	Attack recovery	Data loss protection

3.6.4.2 Alternative patterns for concerns

1) User authentication

#	Pattern	Performance	Security	Difficulty
---	---------	-------------	----------	------------

1	Use session for authentication	M	H	H
2	Use token for authentication	H	H	L

Pattern #2 is selected. In a distributed system, sessions need to be stored on server-side, and synchronization between server nodes is necessary, which brings excessive performance overhead and consistency problem. Tokens, like JWTs(JSON Web Tokens), are more suitable for distributed systems.

2) Input data validation

#	Pattern	Performance	Security	Difficulty
1	Validate data in front-end	H	L	M
2	Validate data in back-end	M	H	M
3	Validate data in both front-end and back-end	M	H	H

Pattern #3 is selected. Validation in front-end can not ensure security at all. However, validation only in back-end leads to excessive requests and high latency, which affects user experience.

3) Encryption method

#	Pattern	Performance	Security	Difficulty
1	Use HTTPS (SSL/TLS) for encryption	M	H	L
2	Use custom encryption protocol	M	H	H

Pattern #1 is selected. HTTPS is widely-used for data encryption, and it ensures transportation layer security. HTTPS is supported by most of browsers, and cost less effort into developing.

4) Data loss protection

#	Pattern	Safety	Cost	Difficulty
1	Backup database regularly	H	M	L
2	Add another database server for real-time copying	H	H	M

Pattern #2 is selected. Pattern #2 is almost real-time, and uses incremental copying method, which brings high performance and high data safety.

3.6.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
User authentication	Use token for authentication	Scenario 3
Input data validation	Validate data in both front-end and back-end	Scenario 4
Encryption method	Use HTTPS (SSL/TLS) for encryption	Scenario 4
Data loss protection	Add another database server for real-time copying	Scenario 12

3.6.5 Element view diagram

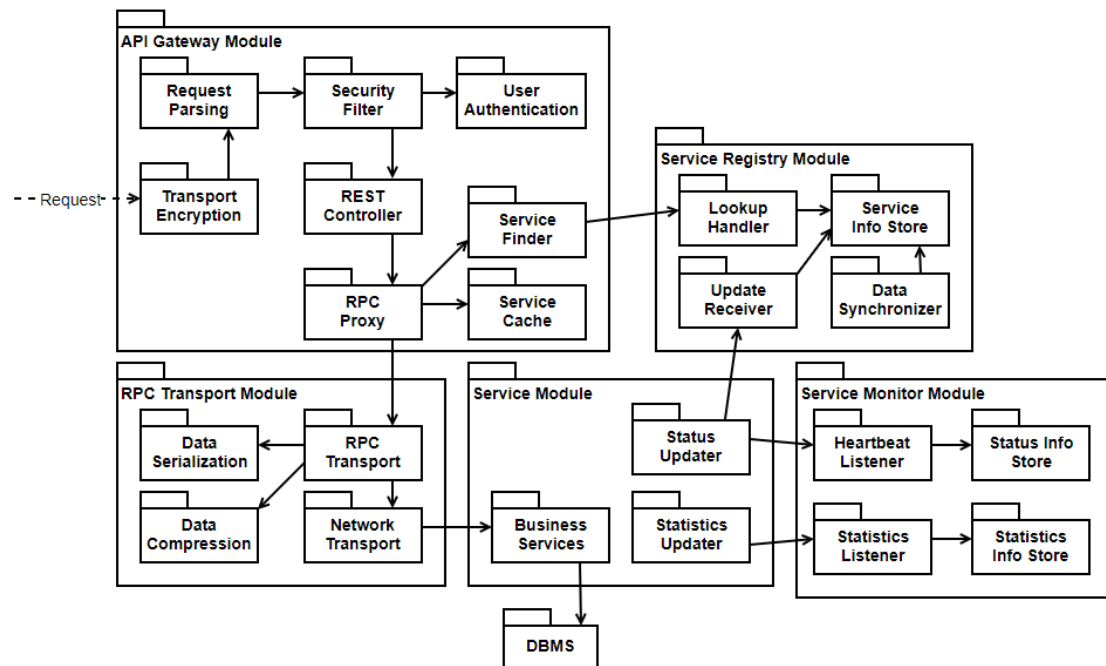
Features of this component are implemented in other modules.
See also 3.4.5 API Gateway element view diagram.

3.6.6 Evaluation

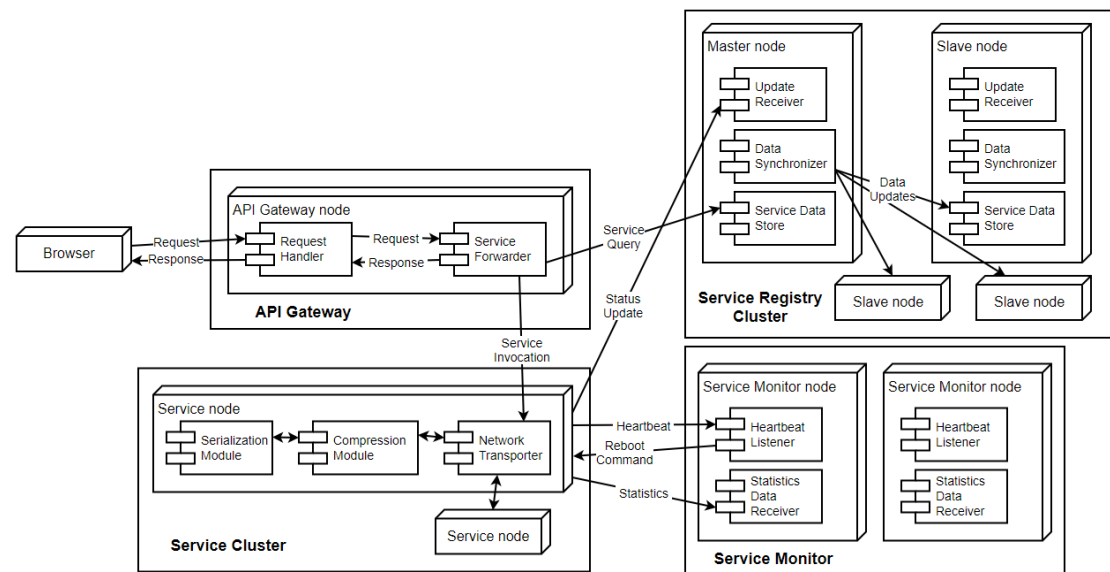
No conflicts were found during the evaluation of this iteration.

3.7 Final architectural views

3.7.1 Module view



3.7.2 Component-Connector view



4. ADD Procedures for MVC Pattern

4.1 Iteration 1

4.1.1 Requirements information

See also 3.1 Iteration 1.

4.1.2 Decomposed system components

Iteration 1 is for the overall architecture design. No system components are decomposed at this iteration.

4.1.3 Element view diagram

See also 2.2 MVC architecture pattern.

4.2 Iteration 2

4.2.1 Requirements information

Requirements information is identical to Iteration 1.

4.2.2 Decomposed system components

The trading module is decomposed at this iteration. This module processes users' orders, matches buy orders and sell orders based on their prices.

4.2.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 1: >1000 users trading at the same time	H	H
Scenario 2: Normal trade operation by user	H	M
Scenario 7: Increasing/decreasing computing power	M	M
Scenario 9: Server down	H	H
Scenario 13: Network outage	M	H

4.2.4 Designing for ASRs

4.2.4.1 Design concerns

Quality Attribute	Design Concern	Subordinate Concerns
Performance	Trade processing speed	Order processing speed
Scalability	Increasing/decreasing trade request processing power	Increasing/decreasing computing power
Reliability	Concurrent operations	Concurrent computing
Reliability	Network outage	Network status checking, Network outage handling

4.2.4.2 Alternative patterns for concerns

1) Order processing speed

#	Pattern	Extensibility	Throughput	Performance	Cost
1	Use queue for order processing	H	H	M	M
2	Upgrade server hardware	M	M	H	H

Pattern #1 is selected. Although pattern #2 improves performance dramatically, it is not a cost-effective solution. Besides, high throughput is more important than performance of a single order in high concurrency situations.

2) Increasing/decreasing computing power

#	Pattern	Impacts	Difficulty	Cost
1	Increase or decrease the count of servers	L	M	M
2	Upgrade server hardware	H	L	H

Pattern #1 is selected. When the hardware of a server is being upgraded, the server must go down first, which will impact existing services.

3) Concurrent computing

#	Pattern	Extensibility	Concurrency	Performance	Cost
1	Add more servers, use load balancing	H	H	H	H
2	Upgrade server hardware	M	L	H	H

Pattern #1 is selected. Concurrency problem is significantly vital in trading systems. These systems must achieve very high concurrency. If we only upgrade server hardware, it is more likely for us to face concurrency bottlenecks. Therefore, pattern #1 is selected though its high cost.

4) Network status checking

#	Pattern	Performance	Time Cost
1	Network outage if not receiving response in a regular interval	H	H
2	Use persistence connection and heartbeat checking	M	L

Pattern #1 is selected. In MVC architecture pattern, it is difficult to maintain persistence connections between servers, and they will consume more system resources. Pattern #1 only needs to handle network outage if responses are not received in a regular interval, thus it consumes less system resources.

5) Network outage handling

#	Pattern	Server Performance	Client Performance
1	Clients retry requests	H	M
2	Clients send acknowledgement	M	M

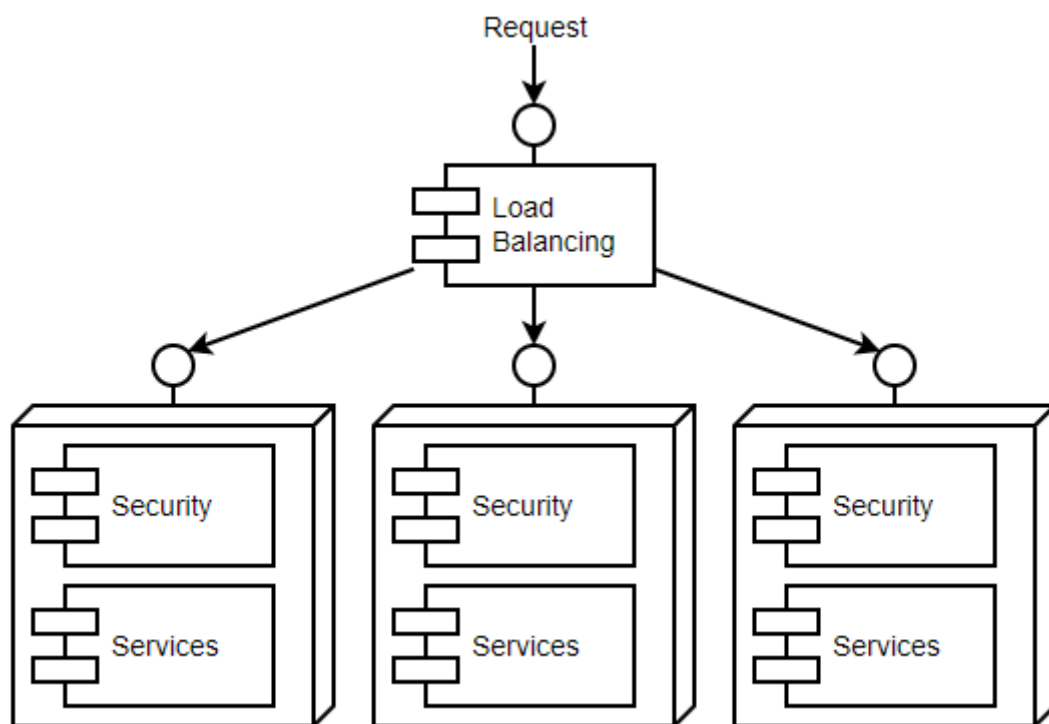
Pattern #1 is selected. Sending acknowledgement consumes more resources on both client side and server side, which also affects performance of other clients.

4.2.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
---------	------------------	-----------------------

Order processing speed	Use pipeline for order processing	Scenario 1, Scenario 2
Increasing/decreasing computing power	Increase or decrease the count of servers	Scenario 7
Concurrent computing	Add more servers, use load balancing	Scenario 1, Scenario 2
Network status checking	Network outage if not receiving response in a regular interval	Scenario 9
Network outage handling	Clients retry requests	Scenario 13

4.2.5 Element view diagram



4.2.6 Evaluation

No conflicts were found during the evaluation of this iteration.

4.3 Iteration 3

4.3.1 Requirements information

Requirements information is identical to Iteration 1.

4.3.2 Decomposed system components

The market information module is decomposed at this iteration. This module is responsible for showing the latest market information to users.

4.3.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 2: Normal trade operation by user	H	H
Scenario 9: Server down	M	L

4.3.4 Designing for ASRs

4.3.4.1 Design concerns

Quality Attribute	Subordinate Concerns
Usability	Clear visualization, easy to use
Portability	Compatible with desktop and mobile browsers
Availability	Show notice when server down

4.3.4.2 Alternative patterns for concerns

1) Clear visualization, easy to use

#	Pattern	Usability	Difficulty
1	Use chart to show prices	H	M
2	Use list to show prices	M	L

Pattern #1 is selected. Charts are the best way to show price trends in a time period, hence the usability of system is higher.

2) Compatible with desktop and mobile browsers

#	Pattern	Portability	Difficulty
1	Use different layouts on desktop and mobile browsers	H	H
2	Use responsive layouts	H	L

Pattern #2 is selected. Responsive layouts can easily fit in different browser sizes without the need to develop more than once.

3) Show notice when server down

#	Pattern	Cost	Difficulty
1	Show notice if not receiving server responses	L	L
2	Actively monitor server status	H	H

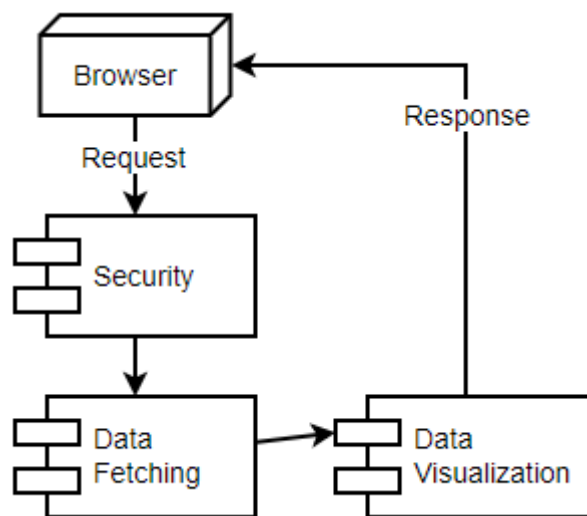
Pattern #1 is selected. In MVC architecture pattern, it will cost much resources for browsers to actively monitor the status of server. Pattern #1 uses passive monitoring, allowing users to know the status of servers without additional overhead.

4.3.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
Clear visualization, easy to use	Use chart to show prices	Scenario 2
Compatible with desktop and mobile browsers	Use responsive layouts	Scenario 2
Show notice when server down	Show notice if not receiving	Scenario 9

	server responses	
--	------------------	--

4.3.5 Element view diagram



4.3.6 Evaluation

No conflicts were found during the evaluation of this iteration.

4.4 Iteration 4

4.4.1 Requirements information

Requirements information is identical to Iteration 1.

4.4.2 Decomposed system components

The data management module is decomposed at this iteration. This module is responsible for insertion, deletion and modification operations of market, order, inventory, and wallet data.

4.4.3 Component-responsible ASRs

Architectural Driver	Importance	Difficulty
Scenario 2: Normal trade operation by user	H	M

4.4.4 Designing for ASRs

4.4.4.1 Design concerns

Quality Attribute	Subordinate Concerns
Usability	Easy to use with clear visualization

4.4.4.2 Alternative patterns for concerns

1) Easy to use with clear visualization

#	Pattern	Usability	Cost
1	Provide all users with a universal set of user interface for buying and	H	M

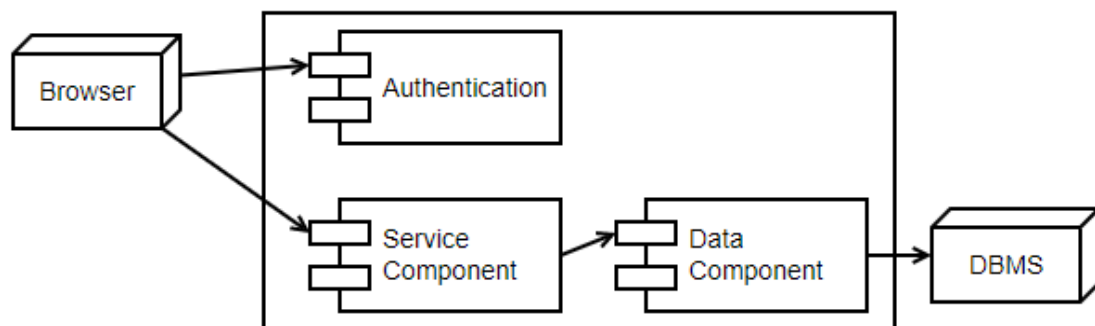
selling		
---------	--	--

Pattern #1 is selected. All users have an identical set of permissions and functions including buying and selling items, since the role of seller in traditional trade model is taken by every user in this system. Therefore, providing universal user interface is the most reasonable solution.

4.4.4.3 Corresponding ASRs of selected patterns

Concern	Selected Pattern	Architectural Drivers
Easy to use with clear visualization	Provide all users with a universal set of user interface for buying and selling	Scenario 2

4.4.5 Element view diagram



4.4.6 Evaluation

No conflicts were found during the evaluation of this iteration.

4.5 Iteration 5

4.5.1 Requirements information

Requirements information is identical to Iteration 1.

4.5.2 Decomposed system components

The security component is decomposed at this iteration. Architectural design of this component is identical to the security component in Microservices pattern.

4.5.3 Component-responsible ASRs

See also 3.6.3 Component-responsible ASRs.

4.5.4 Designing for ASRs

See also 3.6.4 Designing for ASRs.

4.5.5 Element view diagram

Features of this component are implemented in other modules.

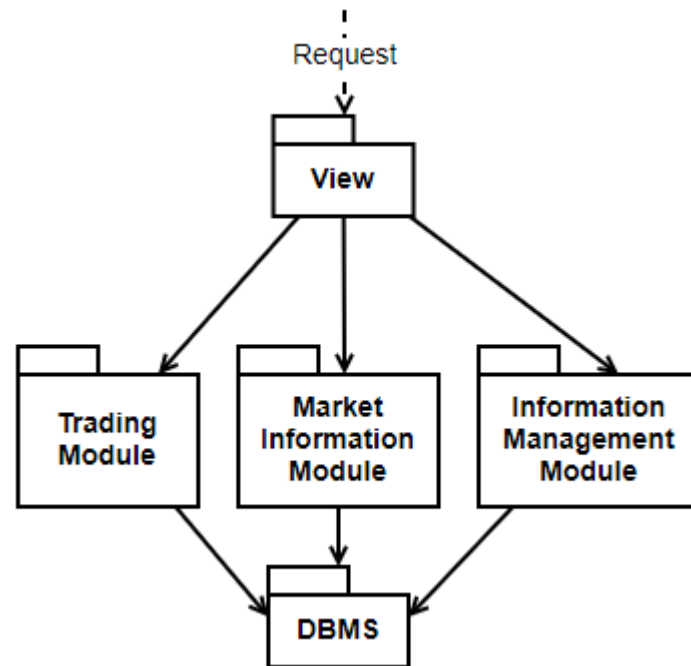
See also 4.2.5 Element view diagram and 4.3.5 Element view diagram.

4.5.6 Evaluation

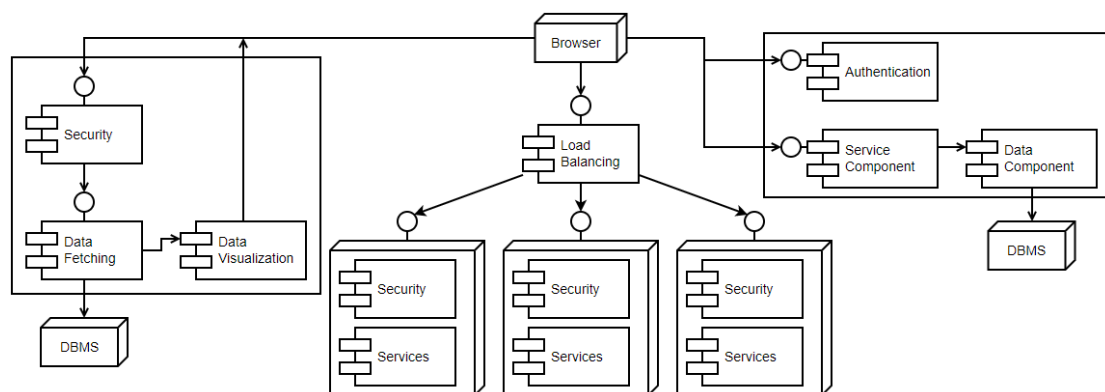
No conflicts were found during the evaluation of this iteration.

4.6 Final architectural views

4.6.1 Module view



4.6.2 Component-Connector view



5. Architecture Patterns Comparison

5.1 Overview

MVC architecture is more favorable for development of small system, providing high performance for low-concurrency scenario, while Microservices architecture shows advantage in high-concurrency scenario and excellent availability. Besides, MVC architecture also allows quicker development, and requires lower costs.

System that employs MVC architecture can be easily maintained and extended. In Microservices architecture, services are deployed independently, which makes it possible for maintainers to add and remove services dynamically, thereby enhancing the scalability and extensibility.

MVC	Microservices
<ul style="list-style-type: none">● Friendly to small system development● High performance at low-concurrency scenario● Easy to maintain and extend● Quick development & lower costs	<ul style="list-style-type: none">● Excellent at high-concurrency scenario● Independent service deployment● Adding/removing services dynamically● Great extensibility and scalability● High availability

5.2 Performance

MVC	Microservices
<ul style="list-style-type: none">● Stable service in low-concurrency and low-traffic environment● High-concurrency performance restrained by data storage and processing capability of a single server	<ul style="list-style-type: none">● Lower speed in low-concurrency scenario due to communication among modules● Quick access in high-concurrency scenario

5.3 Availability

MVC	Microservices
<ul style="list-style-type: none">● Whole system down whenever one server is down	<ul style="list-style-type: none">● Normally one down server won't affect the whole system

5.4 Security

MVC	Microservices
<ul style="list-style-type: none">● Single-direction and highly concentrated data and control flow, easier to ensure security● Attacking one server undermines function of the whole system	<ul style="list-style-type: none">● Great security due to independence of modules and other security measures● Normally attacking one server won't affect the whole system

5.5 Cost

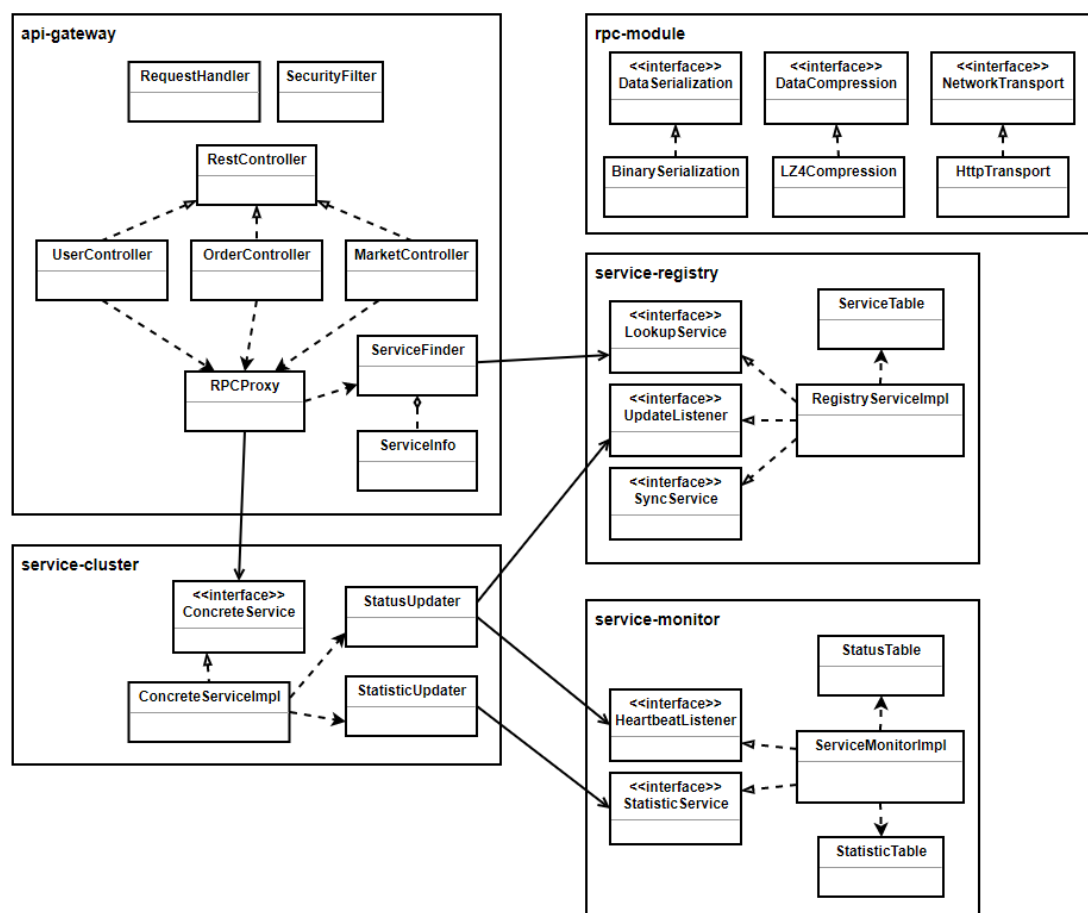
MVC	Microservices
<ul style="list-style-type: none"> • Less servers required, thus lower hardware costs • Lower system complexity, thus lower development costs • Low maintenance costs 	<ul style="list-style-type: none"> • More servers required, thus higher hardware costs • Large data storage necessitated by data backup • Higher system complexity, thus higher development costs • Higher maintenance costs

5.6 Conclusion

Considering the requirement of high concurrency and scalability in this particular system, Microservices pattern is finally chosen.

6. UML Class Diagram and Mapping

6.1 Class Diagram



6.2 Class Mapping

Module	Component	Class
API Gateway	Request handling	RequestHandler, RestController
	Security filtering	SecurityFilter
	Service lookup	ServiceFinder
	Service info cache	ServiceFinder, ServiceInfo
	Service forwarding	RPCProxy
	Load balancing	RPCProxy
Service Registry	Service addition and removal	UpdateListener
	Service lookup handling	LookupService
	Service info store	ServiceTable
	Synchronization between nodes	SyncService
Service Monitor	Heartbeat receiving	HeartbeatListener
	Statistics data receiving	StatisticService
	Statistics data querying	StatisticService
	Service status store	StatusTable
	Service statistics store	StatisticTable
Service Cluster	Status reporting	StatusUpdater
	Statistics reporting	StatisticUpdater
	Business logic	ConcreteService, ConcreteServiceImpl
RPC	Network transport	NetworkTransport
	Data serialization	DataSerialization
	Data compression	DataCompression

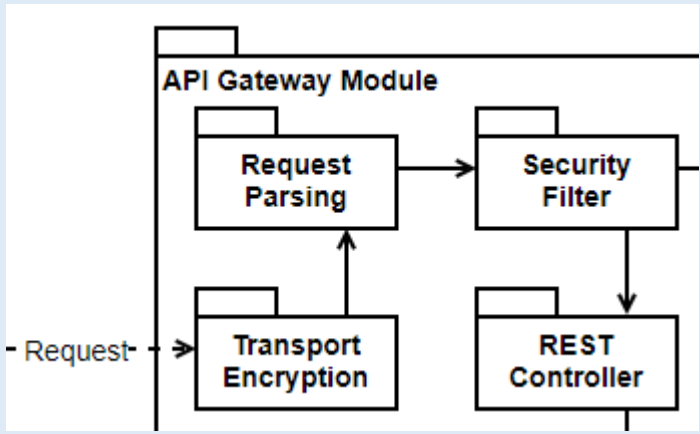
7. ATAM Procedures for Microservices Pattern

7.1 Quality Attribute Utility Tree

Quality Attribute	Concrete Attribute	Scenario
Security	Data encryption	A1: Network traffic wire-tappers cannot decrypt sensitive information [H,L]
	User authentication	A2: System rejects unauthorized requests within 2s [H,M]
Availability	Service availability	A3: Service unavailable time less than 10 minutes per day [H,H]
	Failure detection	A4: Service failures can be detected within 30s [H,H]
	Failure recovery	A5: Recover from failures within 10 minutes [H,M]
Performance	Response time	A6: When 1000 users are trading simultaneously, over 85% requests finish within 1s [H,H]
	Load and capacity	A7: System can handle 1000 simultaneous users [H,H]

Robustness	Input validation	A8: System rejects invalid input within 2s [H,M]
	Error fallback	A9: When error occurs, system responses with fallback data or error message [M,L]
Extensibility	Feature extension	A10: New features or modifications can be deployed within 1h [M,M]
Scalability	System power scaling	A11: Change of computing power is accomplished within 1h [L,M]
Portability	System migration	A12: System migration costs under 2 man-months [L,M]

7.2 ATAM Analysis

Scenario A1: Network traffic wire-tappers cannot decrypt sensitive information					
Attribute	Security				
Environment	System functioning properly				
Stimulus	External network wire-tapping				
Response	Network traffic wire-tappers fail to decrypt sensitive information				
Architectural decisions		Sensitivity	Tradeoff	Risk	Non-risk
Use HTTPS for encryption		S1	T1		N1
Reasoning	This particular system is related to real and virtual properties, thus security is important requirement. Wire-tappers may monitor network traffic, which may lead to sensitive information leakage. Encryption is an effective way to prevent these attacks, though it may slightly affect performance.				
Related Architecture Diagram	 <pre> graph LR subgraph API_Gateway_Module [API Gateway Module] direction LR subgraph TopRow direction LR RP[Request Parsing] --> SF[Security Filter] end subgraph BottomRow direction LR TE[Transport Encryption] --> RC[REST Controller] end RP --> TE SF --> RC end Request[Request] --> TE </pre> <p>The diagram illustrates the internal structure of an API Gateway Module. It contains four main components: Request Parsing, Security Filter, Transport Encryption, and REST Controller. The flow of data is as follows: an external Request enters the module and is first processed by Transport Encryption. The output of Transport Encryption then goes to Request Parsing. From Request Parsing, the data flows to the Security Filter. Finally, the output of the Security Filter is sent to the REST Controller. The entire process is contained within the API Gateway Module boundary.</p>				

Scenario A2: System rejects unauthorized requests within 2s					
Attribute	Security				
Environment	System functioning properly				
Stimulus	Unauthorized user sends a request				
Response	System rejects the request within 2s				
Architectural decisions		Sensitivity	Tradeoff	Risk	Non-risk
Use token for authentication		S2		R1	N2
Reasoning	This particular system is related to real and virtual properties, thus security is important requirement. All trading should be operated by authorized users,				

	token authentication solves this problem effectively without performance impacts.
Related Architecture Diagram	<pre> graph LR Input(()) --> SF[Security Filter] SF --> UA[User Authentication] SF --> RC[REST Controller] RC --> SFinder[Service Finder] </pre>

Scenario A3: Service unavailable time less than 10 minutes per day

Scenario A4: Service failures can be detected within 30s

Scenario A5: Recover from failures within 10 minutes

Attribute	Availability			
Environment	System functioning properly			
Stimulus	A service node is down			
Response	System detects failure within 30s and recover within 10 minutes			
Architectural decisions	Sensitivity	Tradeoff	Risk	Non-risk
Heartbeat monitoring	S3	T2	R2	N3
Try automatic reboot	S4		R3	
Reasoning	Heartbeat monitoring detects service failure faster. Automatic reboot reduces manpower cost and allows quicker recovery.			
Related Architecture Diagram	<pre> graph LR subgraph SM [Service Monitor Module] HU[Heartbeat Updater] --> HL[Heartbeat Listener] HL --> SIS[Status Info Store] STU[Statistics Updater] --> SL[Statistics Listener] SL --> STIS[Statistics Info Store] end HU --> HL STU --> SL </pre>			

Scenario A6: When 1000 users are trading simultaneously, over 85% requests finish within 1s

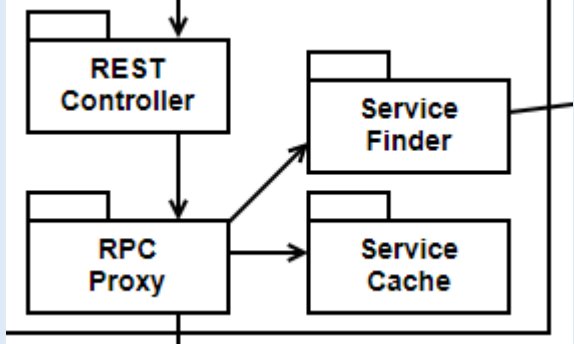
Scenario A7: System can handle 1000 simultaneous users

Attribute	Performance
Environment	System functioning properly
Stimulus	>1000 users trading at the same time
Response	System executes trade operations by users normally Server delivers large-scale requests to computational nodes Data stored in database are modified normally

Architectural decisions	Sensitivity	Tradeoff	Risk	Non-risk
Least response time load balancing	S5			N4
Measure response time of random requests	S5		R4	
Dedicated caching server	S6	T3	R5	N5
Use cross-language binary serialization	S7	T4		N6
Compress data larger than specific size	S7	T4		N6
Reasoning	Performance is also significant in distributed high-concurrency trading systems. Performance is considered and optimized in both API Gateway, RPC protocol and Service Monitor.			
Related Architecture Diagram				

Scenario A8: System rejects invalid input within 2s				
Attribute	Robustness			
Environment	System functioning properly			
Stimulus	User sends a request with invalid input data			
Response	System rejects the request within 2s			
Architectural decisions	Sensitivity	Tradeoff	Risk	Non-risk
Validate data in both front-end and back-end	S8	T5	R6	
Reasoning	Data validation rejects invalid data, which ensures robustness of the whole system.			
Related Architecture Diagram				

Scenario A9: When error occurs, system responses with fallback data or error message	
Attribute	Robustness
Environment	System functioning properly

Stimulus	An error occurred			
Response	System responds with fallback data or error message			
Architectural decisions		Sensitivity	Tradeoff	Risk
Return cached result or error message after an amount of retries (fail-over)		S9		R7
Reasoning	Fail-over mechanism handles error in a way that does not affect user experience.			
Related Architecture Diagram				

Scenario A10: New features or modifications can be deployed within 1h

Scenario A11: Change of computing power is accomplished within 1h

Scenario A12: System migration costs under 2 man-months

Attribute	Extensibility, Scalability, Portability			
Environment	System functioning properly			
Stimulus	System functions need to be extended, changed or migrated.			
Response	New features or modifications can be deployed within 1h Change of computing power is accomplished within 1h System migration costs under 2 man-months			
Architectural decisions		Sensitivity	Tradeoff	Risk
Microservices architecture pattern		S10	T6	N7
Reasoning	This particular system is related to real and virtual properties, thus security is important requirement. Wire-tappers may monitor network traffic, which may lead to sensitive information leakage. Encryption is an effective way to prevent these attacks, though it may slightly affect performance.			
Related Architecture Diagram	See also 2.1 Microservices architecture pattern.			

7.3 Sensitivity points

#	Architectural decision	Reasoning
S1	Use HTTPS for encryption	Sensitivity point for security and performance. Effectively ensures users' information security, but brings performance overhead.
S2	Use token for authentication	Sensitivity point for security and performance. Effectively prevents unauthorized operations.

S3	Heartbeat monitoring	Sensitivity point for availability and performance. Detects service failure faster, but increases network and system resource usage.
S4	Try automatic reboot	Sensitivity point for availability. Allows recovering from service failure faster.
S5	Least response time load balancing Measure response time of random requests	Sensitivity point for performance. Load balancing significantly improves overall performance, while measuring response time slightly affects performance.
S6	Dedicated caching server	Sensitivity point for performance and portability. Caching improves performance, but dedicated caching servers are more difficult to migrate.
S7	Use cross-language binary serialization Compress data larger than specific size	Sensitivity point for performance and portability. Serialization and compression methods have different impacts on performance and different difficulties when migrating.
S8	Validate data in both front-end and back-end	Sensitivity point for robustness and performance. Data validation improves robustness but slightly affects performance on server-side.
S9	Return cached result or error message after an amount of retries (fail-over)	Sensitivity point for robustness. Fail-over mechanism improves robustness of the whole system.
S10	Microservices architecture	Sensitivity point for extensibility, scalability, portability. Services are easy to scale and migrate, but difficult to extend or modify.

7.4 Tradeoff points

#	Architectural decision	Reasoning
T1	Use HTTPS for encryption	Tradeoff point for security and performance. Improves security but decreases performance.
T2	Heartbeat monitoring	Tradeoff point for availability and performance. Improves availability but decreases performance.
T3	Dedicated caching server	Tradeoff point for performance and portability. Improves performance but affects portability. Needs specific operations when migrating.
T4	Use cross-language binary serialization Compress data larger than specific size	Tradeoff point for performance and portability. Improves performance but affects portability.
T5	Validate data in both front-end and back-end	Tradeoff point for robustness and performance. Improves robustness but slightly affects performance.
T6	Microservices architecture	Tradeoff point for extensibility, scalability, portability. Easy to scale and migrate, but difficult to extend or modify.

7.5 Risk

#	Architectural decision	Reasoning
R1	Use token for authentication	Tokens revocation takes more time to become effective than using session for authentication.
R2	Heartbeat monitoring	If network is congested, heartbeat monitoring may be affected.
R3	Try automatic reboot	Requires additional hardware manager support.
R4	Measure response time of random requests	If measurement is not accurate, server usages may be non-uniform.
R5	Dedicated caching server	System performance may be affected when caching server is down
R6	Validate data in both front-end and back-end	User experience may be affected if validation rules are not consistent.
R7	Return cached result or error message after an amount of retries	Cached result may be outdated. The success rate of retrying is often low.

7.6 Non-risk

#	Architectural decision	Reasoning
N1	Use HTTPS for encryption	HTTPS is supported by most web servers and browsers.
N2	Use token for authentication	Mature solutions like JSON Web Token.
N3	Heartbeat monitoring	Heartbeat monitoring is widely-used in distributed systems.
N4	Least response time load balancing	Load balancing is widely-used in distributed systems.
N5	Dedicated caching server	Mature solutions like Memcached, Redis, ...
N6	Use cross-language binary serialization Compress data larger than specific size	Cross-platform methods (e.g. Hessian) are widely-used.
N7	Microservices architecture pattern	Microservices architecture pattern is widely-used and proven practicable.

8. Summary

8.1 Challenges and experience

At the beginning of this project, our group members generally had little understanding of distributed system architecture. Therefore, we encountered many problems when designing, such as the synchronization methods between different nodes. With the help of various reference materials as well as existing distributed systems, we are finally able to analyze different patterns for distributed system architecture.

While designing, we placed great emphasis on ADD methods. ADD focuses on quality attributes, and is totally different from conventional designing methods focusing on system functions. It helps us find a balance among many quality attributes by comparing different patterns with ASRs.

As a group of 8 students, we are divided into 3 subgroups: 2 students work on Microservices pattern, 4 of us work on MVC pattern and others evaluate the design. This division allows us to work more efficiently.

8.2 Individual's contributions

Name and ID	Contributions
梁家铭(151250091)	Designing Microservices architectural pattern (Iteration 2, 3, 4, 5) Project presentation and demo developing Document integration and translation
李一然(151250087)	Identifying ASRs and scenarios of the project Comparing two architectural patterns Document integration and translation
倪辰皓(141250096)	Designing MVC architectural pattern (Iteration 1, 2)
曹鸿荣(151250006)	Designing MVC architectural pattern (Iteration 3)
常德隆(151250011)	Designing MVC architectural pattern (Iteration 3, 4) Drawing view diagrams
陈进(151250013)	Designing Microservices architectural pattern (Iteration 6) Comparing two architectural patterns
陈淼(151250015)	Searching and reading related materials Evaluating Microservices architectural pattern using ATAM methods
陈锐(151250016)	Searching and reading related materials Evaluating Microservices architectural pattern using ATAM methods