Team Name: A, 你在哪里?

Team Members:
Chang Ern Rae, Chang Jian Ming, Lee Wei Da, Tan Rong Jian Mark, Tan Jun Hui

Background and Business Needs:
SMU IT! started as a brick and mortar IT sales store in SMU campus, serving students primarily. With the rise of e-commerce, SMU IT! is looking to branch into the online market to diversify their customers' portfolio. Therefore, SMU IT!, an online e-store to allow the public to search and purchase the IT products it is selling, is being initiated.

It allows online payment through STRIPE. Users are allowed to give their review on the product. Admin users are allowed to manage the inventory and orders.

Stakeholders

Stakeholder	**Stakeholder Description**
Website Administrators (SMU IT! Store owners)	Web Administrators are responsible for managing the inventory of the store and orders.
Customer (users)	Public users who purchase IT products from the web application. They mainly consist of SMU students.

Key Use Cases:

Use Case Title - Pay for items in shopping cart	
Use Case ID	1
Description	Users retrieve the items in the cart that are stored in the database session state and confirms payment with his or her credit card credentials. System integrates with Stripe payment API using Stripe's Java library through HTTPS protocol to validate the user's credit card credentials and completes payment. Upon successful payment, the system will update the database of a successful purchase and items in cart stored in the in-memory cache will be cleared. With an unsuccessful payment, items in cart stored in database session state will be retained and an unsuccessful message will be displayed for the user. **Alternate flow**: There is a secondary database on hot standby that is mirroring the primary database. If the primary database is down, the secondary database will be updated with the successful purchase instead. Future updates will be done on the secondary database until recovery of primary database. (Master-slave database set up)

Actors	Customers
Main Flow of events	1. Customer selects Stripe and clicks checkout button from payment.jsp 2. Stripe payment window popup 3. Customers key in credit card details and click the pay button 4. System invokes Stripe Payment API through HTTPS Protocol 5. Upon successful payment, system updates the database on the successful purchase. 6. System removes items in cart stored in the database session state 7. Customer is shown a successful purchase message
Alternative Flow of events	1. Upon successful payment, the system attempts to update the primary database on the successful purchase but primary database is down 2. System will update the secondary database that is on warm standby on the successful purchase instead 3. With successful update, system removes items in cart stored in the database session state 4. Customer is shown a successful purchase message
Pre-conditions	• At least 1 item in cart stored in database session state • Stripe payment API and web servers are working
Post-conditions	• Success / Error status

Use Case Title - Access website with 300 concurrent users and users are able to view website within 5 seconds under normal circumstances where all parts of the system, e.g. load balancer, is working (Performance: Response within 5 seconds) (Availability: Throughput)	
Use Case ID	2
Description	300 concurrent customer attempts to access the product web page at a single point in time. The requests will be connected to the reverse proxy/load balancer where they will be diverted using round robin scheduling into one of the two identical tomcat instances in their first connection, to display the webpage to the users. User will be recognised by IP (IP Hash) to implement sticky session so that user will continue to interact with the same tomcat instance for subsequent

	requests. Each of the tomcat instance will make a request to the database and display the available items for purchase to the customer. Time taken for any customer to see the website should be within 5 seconds.
Actors	Customers
Main Flow of events	1. 300 concurrent customers access the website 2. The 300 requests will be sent to the reverse proxy/load balancer 3. Request will be served based on round robin scheduling evenly into one of the two identical tomcat instances 4. Tomcat instance will make a request to the database to retrieve the available items and display to the customer 5. Subsequent request by a user will be made to the same tomcat instance 6. A customer will take 5 seconds or less to see the website
Alternative Flow of events	-
Pre-conditions	• Reverse Proxy / Load balancers, both tomcat instances and database are running
Post-conditions	• User will be able to display the website on their browser • Future requests in the same session will be routed to the same tomcat instance

Use Case Title - Access and view website 99.9% of a day (24 hours) (Availability: Uptime)	
Use Case ID	3

Description	In a day (24 hours), a customer is able to access and view the website 99.9% of the time. The request will be connected to the reverse proxy/load balancer where they will be diverted using round robin scheduling into one of the two identical tomcat instances in their first connection, to display the webpage to the users. User will be recognised by IP (IP Hash) to implement sticky session so that user will continue to interact with the same tomcat instance for subsequent requests. Each of the tomcat instance will make a request to the database and display the available items for purchase to the customer. **Alternate flow 1:** The 2 tomcat instances serves as fault tolerance where in the situation where one fails, all requests will be channelled to the other running tomcat instance. **Alternate flow 2:** Passive load balancer/reverse proxy serves as fault tolerance where in the situation where the active main load balancer fails, passive load balancer/reverse proxy will take over the processing of requests. Master-slave database is required to ensure no single point of failure in the set up. This is discussed in earlier section in use case 1.
Actors	Customers
Main Flow of events	1. Customer connects to the webpage 2. The request will be sent to the reverse proxy/load balancer 3. Request will be served based on ip hash (sticky session) and round robin scheduling into one of the two identical tomcat instances 4. The tomcat instance will make a request to the database to retrieve the available items and display to the customer
Alternative Flow of events 1	1. Customer connects to the webpage 2. The request will be sent to the reverse proxy/load balancer 3. Request is served based on round robin scheduling into one of the two identical tomcat instances 4. If one tomcat instance is down, all request will be sent to the other tomcat instance 5. The tomcat instance will make a request to the database to retrieve the available items and display to the customer

Alternative Flow of events 2	1. Customer connects to the webpage 2. The request will be sent to the passive reverse proxy/load balancer when active reverse proxy/load balancer is down 3. Request is served based on round robin scheduling into one of the two identical tomcat instances 4. The tomcat instance will make a request to the database to retrieve the available items and display to the customer
Pre-conditions	• Active and passive Reverse Proxy / Load balancers, both tomcat instances and master and slave database are running
Post-conditions	• User will be able to access the website on their browser

Use Case Title - Only authorised users (admins) can have access to the database and have the ability to write to the database (Security: Database confidentiality and integrity)	
Use Case ID	4
Description	Wanting to make any changes in the database (e.g. updating pricing of items sold), admins will log in with their authorised log-in credentials. After submission of the log-in credentials, the system will connect to the database to authenticate the user. Unauthorised users will not be given access to the server by the system. **Alternative flow**: User tries to access or edit the database indirectly through sql injection. Database is hardened to prevent sql injection to maintain the integrity of the data.
Actors	Admins
Main Flow of events	1. Admin logs in using their username and password 2. System connects to the database to authenticate the username and password entered 3. Authorised set of log in credentials will allow user to access and edit database whereas unauthorised set of log in credentials will not allow user to access database
Alternative Flow of events	1. Access and edit database using sql injection 2. Database is hardened to reject any sql injection
Pre-conditions	• No known security exploits with technology used
Post-conditions	-

Use Case Title - Unauthorised injection of malicious script (Security: Server confidentiality and integrity)	
Use Case ID	5
Description	Users will be able to make comments through the comments section of the product. However, users with malicious intent will not be able to inject malicious scripts into the server system to steal information or perform other malicious act to the server
Actors	User
Main Flow of events	Malicious user will try to inject a malicious script into the server through comments section Malicious script will be rejected from the system
Alternative Flow of events	-
Pre-conditions	• No known security exploits with technology used
Post-conditions	

Key Architectural Decisions

Architectural Decision - Load Balancing/Reverse Proxy between 2 Web Application Tomcat server	
ID	1
Issue	Single point of failure for web application tomcat server.
Architectural Decision	2 Tomcat web servers will run a copy of the SMU IT application each. Load Balancer/Reverse Proxy, Nginx, will be used to receive client's requests and routed into the 2 servers depending on the availability of each server. While increasing availability, it increases performance as well, since the load is being shared between 2 web servers by Nginx using round robin. For subsequent connections, a user will be recognised by their IP address and Nginx will send that user request to the same server unless the server is unavailable.
Assumptions	-

Alternative	API Gateway with microservices
Justification	1. Our application has a linear business where user logs in, add item and checks out. Breaking down this application into microservices will increase the hardware needed to achieve the same availability as the set up mentioned above as we will have to ensure that none of the microservices become a single point of failure. Also, if any of the microservice is a single point of failure, if that microservice becomes unavailable, it will render the entire application useless as the linear business flow is broken. With a linear business set up that has no need for exposing certain microservice for external usage and does not require any individual functionality to have higher performance than another (no need to scale one services to have higher performance than another), breaking down into microservices only increase cost without bringing any benefits. 2. Breaking an application into microservices will decrease performance where more network latency will be introduced into the system as calls to each microservice will have to be made to allow the application to run.

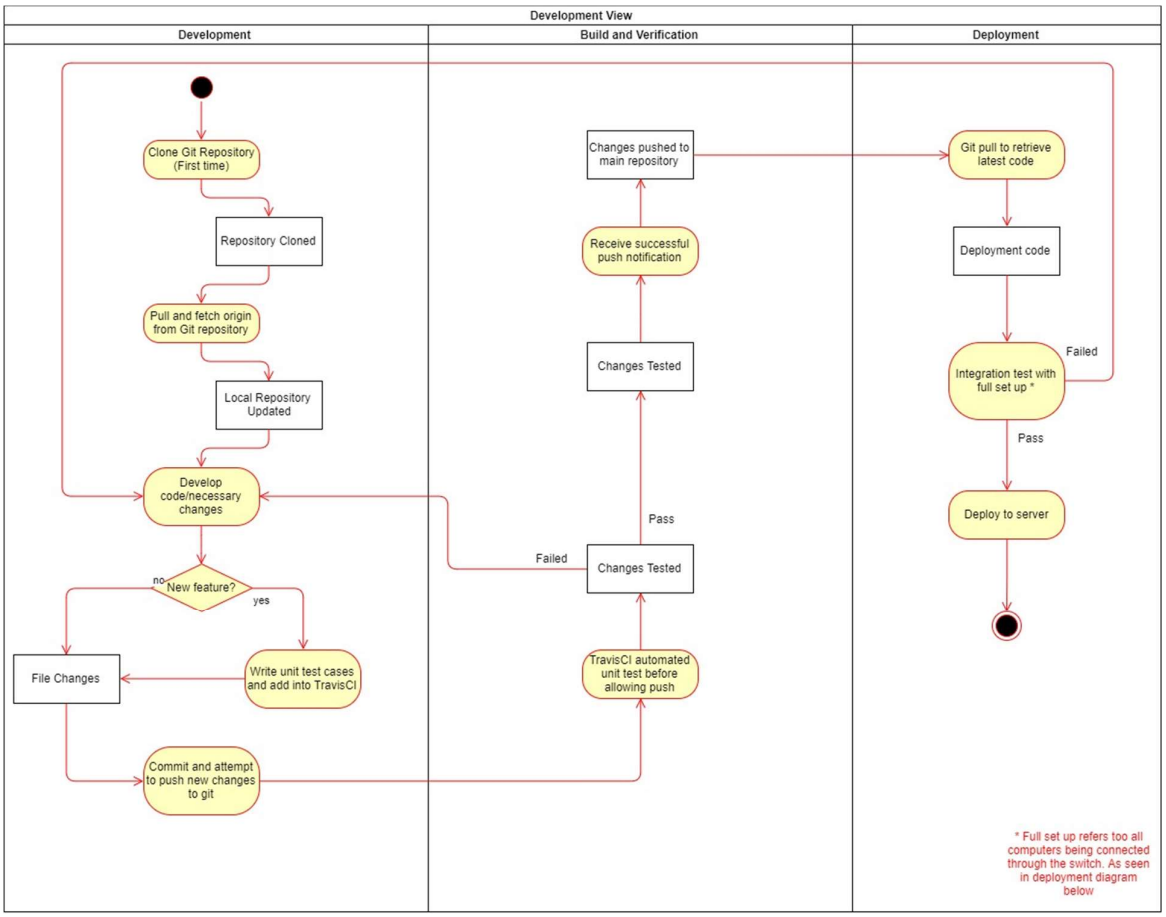
Architectural Decision - Session Replication among 2 tomcat servers	
ID	2
Issue	In the instance of failover between the 2 tomcat servers, user information is not replicated among the 2 servers and user will have to for instance, re-log in or have different or empty cart.
Architectural Decision	Session replication among the 2 tomcat servers is done. There are 2 main functionalities that require session replication: (1)User log-in credentials which is accomplished through storing user session in database with HttpSession and Tomcat's Persistent Manager using JDBC Based Store. User credentials is not stored at the client side as it can be a security issue. (2) Cart details are stored and retrieved manually from the database through SQL calls. This allows user to access saved cart items between different servers or if any one server is down.
Assumptions	-
Alternative	-
Justification	-

Architectural Decision - Active-passive load balancer/reverse proxy	
ID	3
Issue	While increasing performance and availability by using Nginx to serve request across the 2 web application servers, the server hosting Nginx becomes a single point of failure in the entire set up
Architectural Decision	A passive load balancer/reverse proxy is used. A batch script is used to ping main computer with active Nginx. If active Nginx fails to respond to passive Nginx ping, passive Nginx will take over the active Nginx IP address and continue serving the requests. Passive and active Nginx will have the same configuration.
Assumptions	-
Alternative	Using Keepalived to achieve active-active load balancing cluster
Justification	Keepalived is a tool that is used with nginx-plus to achieve active active load balancing cluster which gives an additional benefit of being able to handle high capacity loads since there are 2 active load balancers serving the request. However, keepalived is only compatible to Linux machines and due to hardware limitations, we can only go with an active passive set up, forgoing the additional benefit active active load balancing that keepalived brings. In an ideal scenario where we are not constrained by hardware limitations in the project, we will use an active active load balancing cluster set up

Architectural Decision - Master-slave database	
ID	4
Issue	2 Tomcat web application servers will read and write into a single database and that database server is a single point of failure in the set up
Architectural Decision	Slave database will be set up as a hot standby, replicating the data in the master database server constantly whenever there are changes in the master database. In the situation where master database server is down, web application server will read and write into the slave database. Slave database will not replicate master database once master database is down and replication will only take place after manual reset of the master-slave replication. Manual reset is used so that we can ensure there are no data conflict or loss since these data

	contain transaction and purchase records which is critical in the business context.
Assumptions	Master database will not automatically reconnect back into the network after failure.
Alternative	Master-master database
Justification	We want to avoid instances where we can receive data updates at the same time which results in data conflict. Manual intervention will be required for the data conflict and it could potentially be time consuming especially for an ecommerce store. Therefore, we went for a centralised database setup and used master-slave replication to avoid having the database as a single point of failure.

Development View

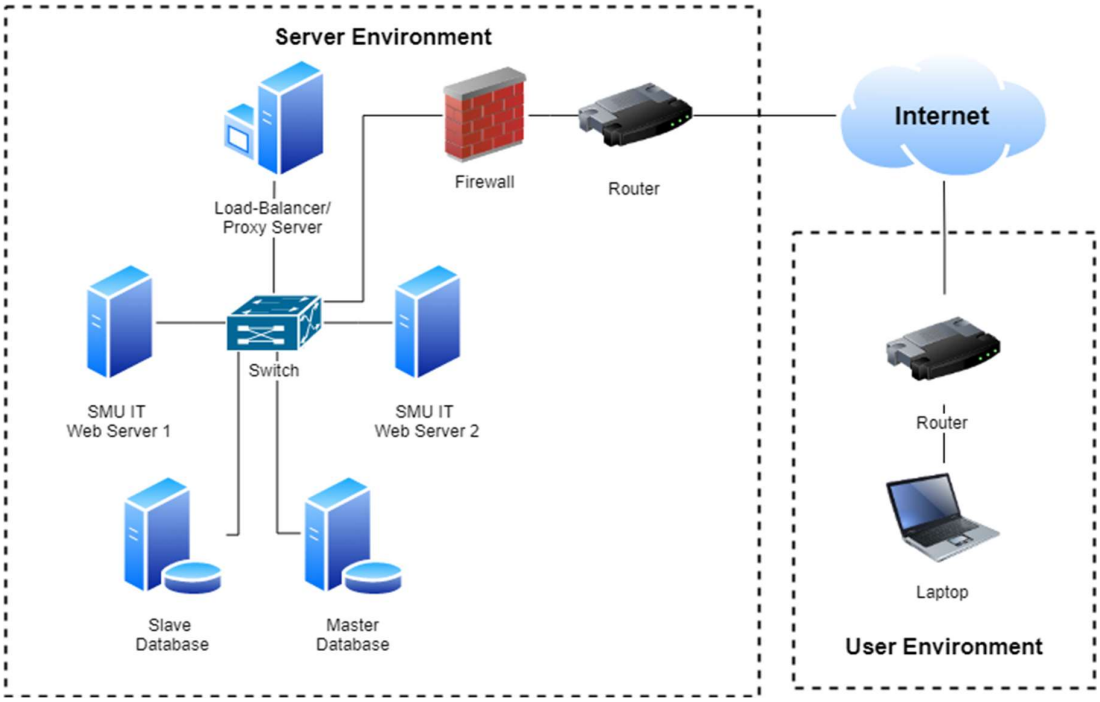


- **Notes:**
  - TravisCI help to automate unit test
    - Unit test is done using Junit

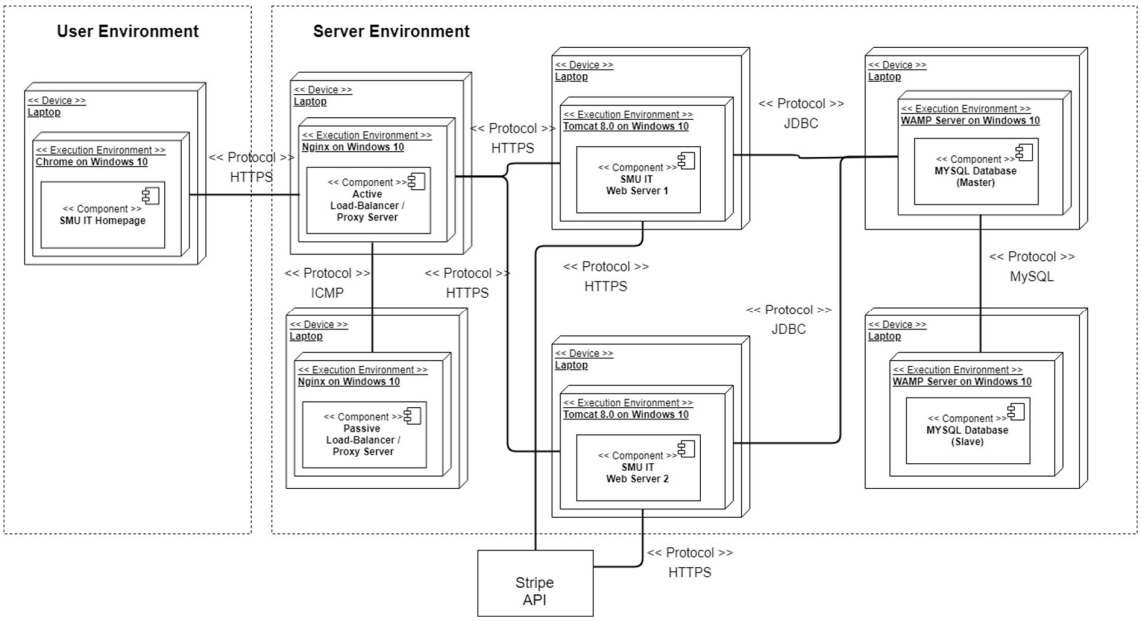
- Integration test is conducted with integration test that is done with Katalon. Integration test can only be performed when the computer used to conduct the integration test is connected to another computer with the database running. This is due to the configurations made in the tomcat server and application configurations
- Deployment to production environment is done manually as we are using local machine set ups and it is impossible to deploy automatically like if we are using cloud services

Solution View

Network Diagram



Deployment Diagram



- To achieve maintainability, we focused on 2 key aspects (1) code-based modularity (Software) and (2) architectural style (Hardware)
    (1) Code-based modularity is achieved through design patterns and breaking down the code in different modules. For instance, database connection is done using the Singleton pattern design with the SQLCallSingleton file. Since many files in the project requires connection to database, they get the instance of the singleton and then use the getConnection() method. Any changes that need to be made to the connection configuration (eg. host, username, or password) can be made to SQLCallSingleton class without affecting other files.

    With code-based modularity, it increases testability also as unit test can be conducted on each module to ensure that a particular module is working as defined before integrating all modules together.

    Design patterns used: Proxy, Singleton, Façade and Builder

    (2) Our architectural style achieves maintainability as each key component has it back-up alternative. For instance, if load balancer configurations need to be changed, changes can be deployed to the passive load balancer first and passive load balancer can take over active load balancer while changes are being deployed to the active load balancer. Essentially, increasing the modifiability on a hardware level.
- Alternative: We considered breaking the application into microservices as that is the first idea many will get to achieve modularity. Refer to key architectural decision 1 justification for the reason.

Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Load balancer/reverse proxy (Nginx)	Tomcat web application server	HTTPS	HTTP Message	Synchronous
Tomcat web application server	Database	HTTPS	HTTP Message	Synchronous
Master database	Slave database	MySQL	Binary Log	Asynchronous
Passive load balancer/reverse proxy	Active load balancer/reverse proxy	ICMP	ICMP Payload	Synchronous

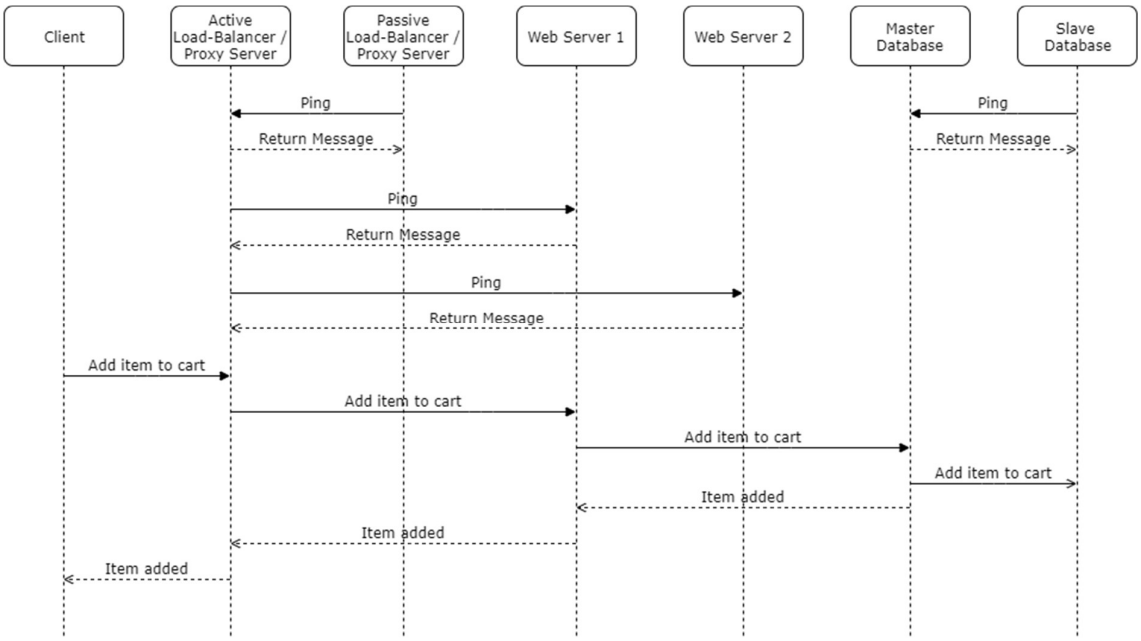
Hardware/Software/Services Required

No	Item	Quantity	License	Buy/Lease	Cost
1	Linux server	6	-	Buy	$4000
2	Router	1	-	Buy	$400
3	High performance switch	1	-	Buy	$1000
4	Nginx-plus	2	Proprietary	Buy for the support	$2000 per annum
5	MySQL enterprise edition	2	Proprietary	Buy for the support	$5000 per annum
6	Apache TomEE	2	Proprietary	Buy for the support	Unknown (Contact to know pricing)
7	Katalon	1	Proprietary but free	-	$0
8	Stripe Payment Service	1	Proprietary	Buy	3.4% + $0.50 per successful card charge

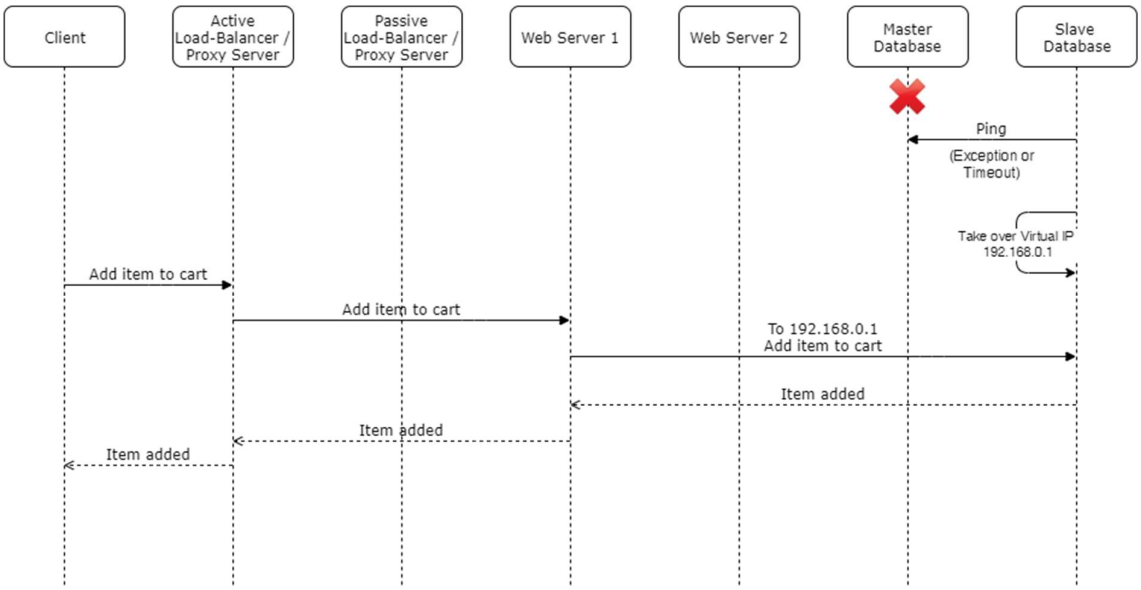
Availability View

Node	Redundancy	Clustering			Replication			
		Node Config.	Failure Detection	Failover	Repl. Type	Session State Storage	DB Repl. Config.	Repl. Mode
Nginx	Horizontal	Active-passive	Ping	Virtual IP	-	-	-	-
Database	Horizontal	Active-passive	Ping	Virtual IP	-	-	Master-Slave	Asynchronous
Web server	Horizontal	Active-active	Ping	Load balancer	-	Client and database	Master-Master	-

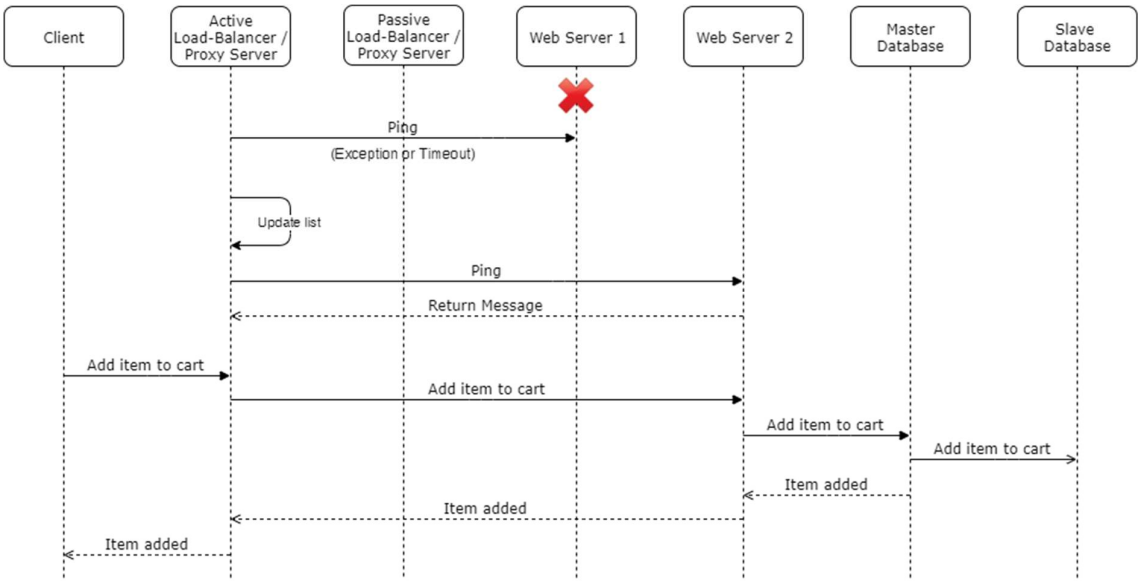
Sequence Diagram: All Services Working



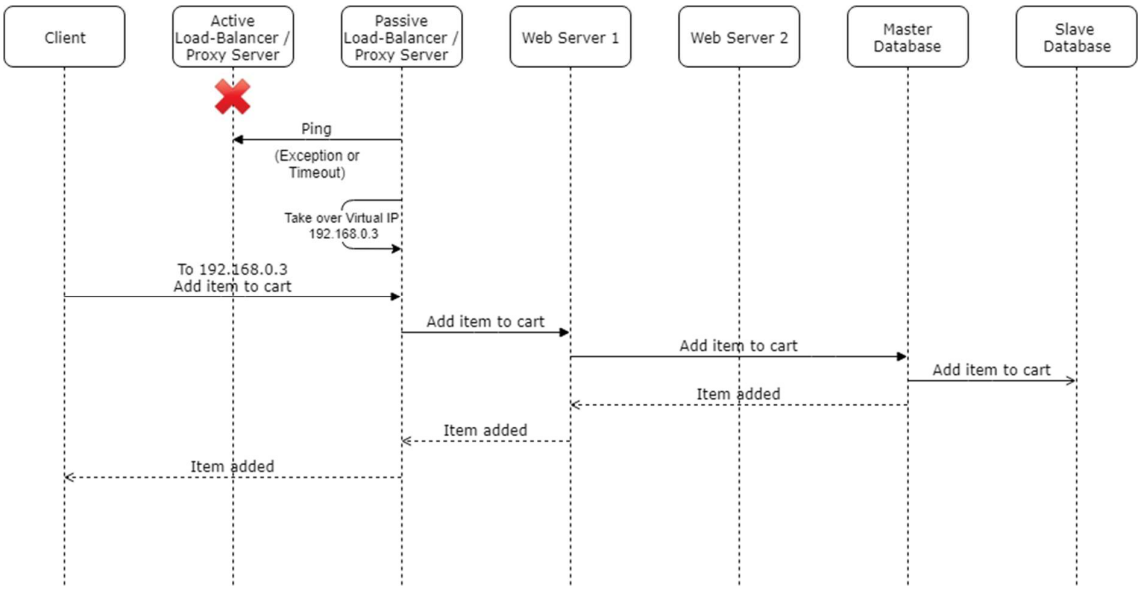
Fail-Over: Master Database Down



Fail-Over: Web Server 1 Down



Fail-Over: Active Load-Balancer / Proxy Server Down



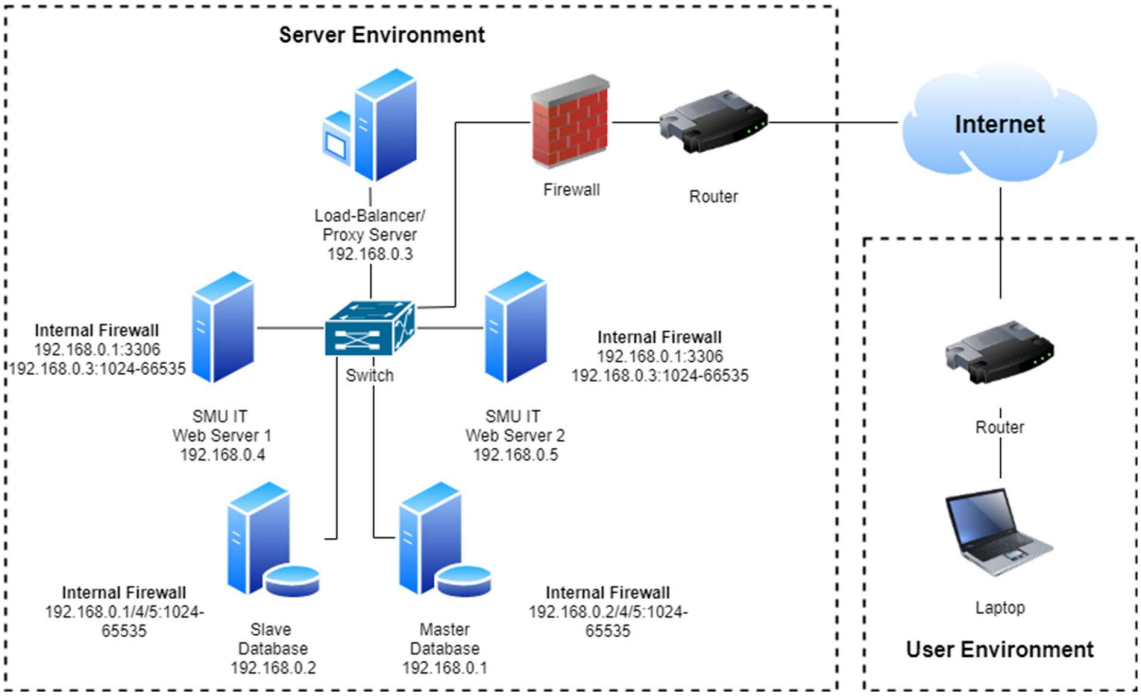
Security View

No	Asset/Asset Group	Potential Threat/Vulnerability pair	Possible Mitigation Control
1	User account details in database	SQL injection - No validation of input and user password is not hashed. Confidentiality is compromised.	Validate input and store hashed password
2	Login credential details when logging in	Man-In-The-Middle - No encryption when transporting data over network. Confidentiality is compromised.	SSL encryption of data
3	Client sensitive data such as credit card details when interacting with our website	Cross Site Scripting (XSS) - No validation of user input and lack of content security policy usage. Confidentiality and integrity are compromised	Validate user input to prevent malicious user input and adding XSS protection headers in system configurations such as nginx
4	Web server and database server integrity	Malware - No control of data transfer or access into the web and database server. Confidentiality, integrity or	Use firewall to prevent external access or data transfer into web and database server. For instance web server can only transfer data with database server and proxy.

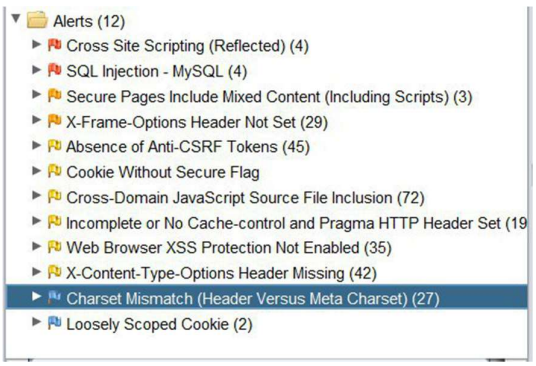
		availability can be compromised depending on type of malware	Network is isolated from the internet and only proxy can communicate with the internet. Malware and malicious script detection in proxy server to prevent introduction of malware into the private network group
5	URL pages with sensitive data	Unrestricted URL Access - Attackers try to gain access to hidden privilege pages/locations/resources. Confidentiality is compromised	Validate session and access rights of user on every page and resource

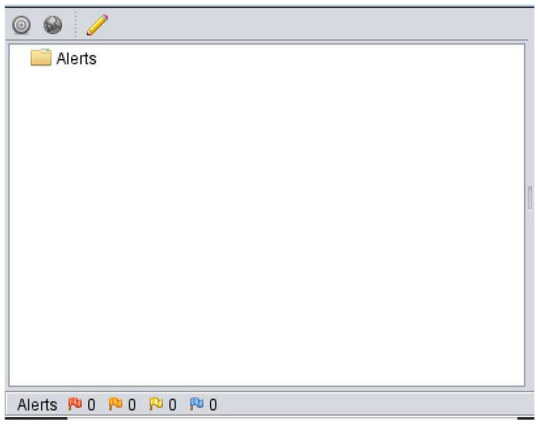
- **Note: All known security vulnerabilities are resolved**

Firewall settings:



OWASP ZAP	
Before	**After**





Performance View

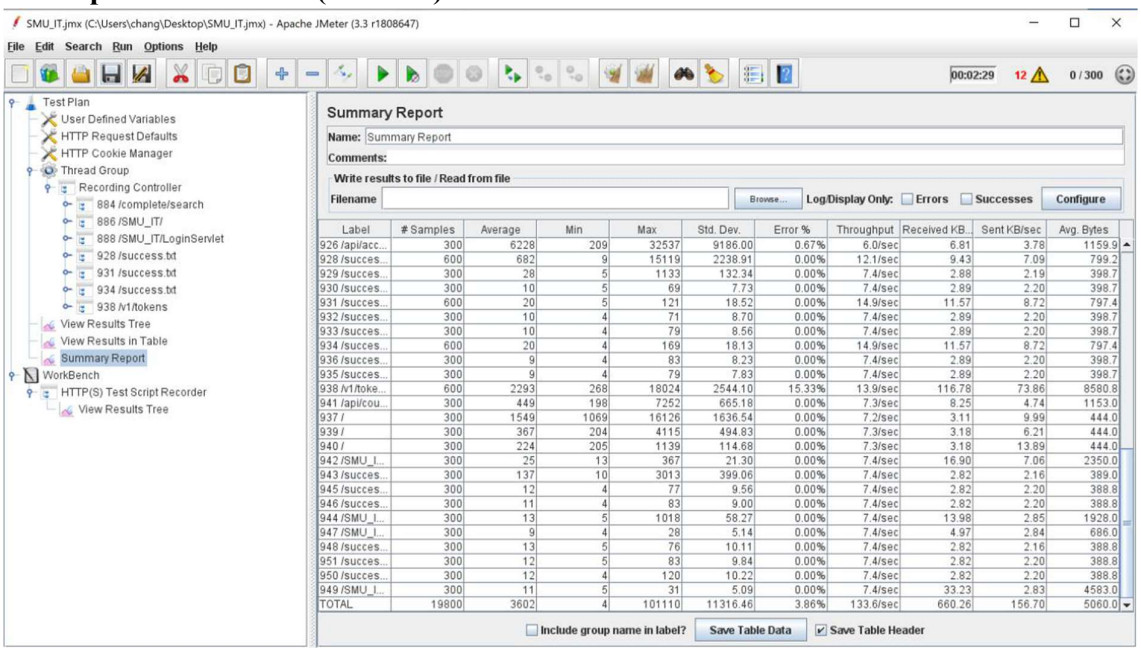
No	Description of the Strategy	Justification
1	Load Balancing - Implemented Incoming requests will be directed across the 2 tomcat web servers using nginx as a load balancer. This spreads the workload across the 2 servers instead of one, increasing the capacity of the entire solution.	Load balancing can be more effective with increasing number of web application servers. However, due to resource limitation, we will be using 2 tomcat servers.
2	Gzip - Implemented One of the factors affected data transferred from web server to client is data size. Gzip compress the data before passing it through the network to the client which lowers the amount of data transfer, reducing processing and response time.	An alternative way to reduce data being transferred is image size optimisation by reducing the size of image. However, as we are using external images from the respective companies through URL, e.g. iPad image from Apple website, we are unable to control the size of the image. Therefore, Gzip is used to reduce data size instead.
3	Link Prefetch - Implemented Prefetch is used to cache resources that will certainly be used by the web browser. This increases the user experience by decreasing the time taken to render a page.	-

	The browser will begin requesting for images from its respective URLs after the page is done loading. Images will be downloaded and stored in the browser cache where it can be easily and quickly accessed when needed. As an ecommerce site, apart from making purchases, the site will mainly be used for browsing of products (i.e. resources like product images will definitely be used), thus low latency when browsing between product pages is crucial. Therefore, prefetching the images and logos will enable a smoother transition between pages since the resources are not requested every time a new page is accessed.	
4	Caching - Implemented Caching is used to increase the performance of data retrieval by reducing the need to access the storage layer. Client-side caching was implemented to reduce the latency involved when requesting web resources from websites.	Caching can be applied on multiple layers like Client-side, DNS, Web, Application and Database layer. Alternative methods like database caching was not implemented after considering the business needs and scale of the ecommerce store. An external cache could decrease the availability of the application as caches do not have high availability properties and can easily fail or result in partial failures. This may affect the consistency of the data when the browser retrieves data from the cache, which does not fulfil the requirements of an Ecommerce store - high availability and low latency.
5	Parallel Execution - Not Implemented Tasks or processes are broken down and split into multiple smaller tasks that do parts of the work at the same time. This reduces the total processing time required by each task as it makes use of the multiple CPU and I/O	-

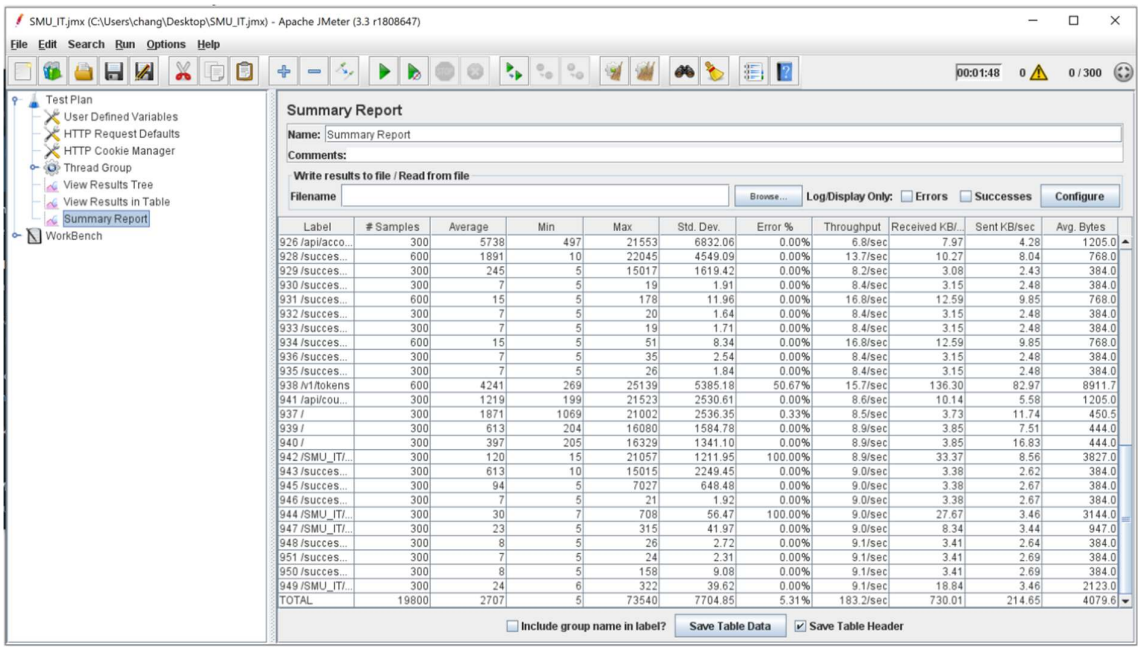
	resources that are typically found in servers today	

- Performance test was conducted using JMeter before all changes and after all the above-mentioned changes as some changes only provide small increment in performance level and is not significant enough to be shown in every strategy. Overall, through what was implemented, we achieved a 1 second improvement for 300 concurrent users.

Initial performance test (Baseline):



Final performance test:



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
926 /api/acco...	300	5738	497	21553	6832.06	0.00%	6.8/sec	7.97	4.28	1205.0
928 /succes...	600	1891	10	22045	4549.09	0.00%	13.7/sec	10.27	8.04	768.0
929 /succes...	300	245	5	15017	1619.42	0.00%	8.2/sec	3.08	2.43	384.0
930 /succes...	300	7	5	19	1.91	0.00%	8.4/sec	3.15	2.48	384.0
931 /succes...	600	15	5	178	11.96	0.00%	16.8/sec	12.59	9.85	768.0
932 /succes...	300	7	5	20	1.64	0.00%	8.4/sec	3.15	2.48	384.0
933 /succes...	300	7	5	19	1.71	0.00%	8.4/sec	3.15	2.48	384.0
934 /succes...	600	15	5	51	8.34	0.00%	16.8/sec	12.59	9.85	768.0
936 /succes...	300	7	5	35	2.54	0.00%	8.4/sec	3.15	2.48	384.0
935 /succes...	300	7	5	26	1.84	0.00%	8.4/sec	3.15	2.48	384.0
938 /v1/tokens	600	4241	269	25139	5385.18	50.67%	15.7/sec	136.30	82.97	8911.7
941 /api/cou...	300	1219	199	21523	2530.61	0.00%	8.6/sec	10.14	5.58	1205.0
937 /	300	1871	1069	21002	2536.35	0.33%	8.5/sec	3.73	11.74	450.5
939 /	300	613	204	16080	1584.78	0.00%	8.9/sec	3.85	7.51	444.0
940 /	300	397	205	16329	1341.10	0.00%	8.9/sec	3.85	16.83	444.0
942 /SMU_IT/...	300	120	15	21057	1211.95	100.00%	8.9/sec	33.37	8.56	3827.0
943 /succes...	300	613	10	15015	2249.45	0.00%	9.0/sec	3.38	2.62	384.0
945 /succes...	300	94	5	7027	648.48	0.00%	9.0/sec	3.38	2.67	384.0
946 /succes...	300	7	5	21	1.92	0.00%	9.0/sec	3.38	2.67	384.0
944 /SMU_IT/...	300	30	7	708	56.47	100.00%	9.0/sec	27.67	3.46	3144.0
947 /SMU_IT/...	300	23	5	315	41.97	0.00%	9.0/sec	8.34	3.44	947.0
948 /succes...	300	8	5	26	2.72	0.00%	9.1/sec	3.41	2.64	384.0
951 /succes...	300	7	5	24	2.31	0.00%	9.1/sec	3.41	2.69	384.0
950 /succes...	300	8	5	158	9.08	0.00%	9.1/sec	3.41	2.69	384.0
949 /SMU_IT/...	300	24	6	322	39.62	0.00%	9.1/sec	18.84	3.46	2123.0
TOTAL	19800	2707	5	73540	7704.85	5.31%	183.2/sec	730.01	214.65	4079.6

Overview of Gzip and Caching: