IMPLEMENTATION OF A WEB-BASED AUDIENCE RESPONSE SYSTEM AS MICROSERVICE APPLICATION VS. MONOLITHIC APPLICATION

Iris Braun, Manuel Hoffmann and Robert Mörseburg

Faculty of Computer Science, Technische Universität Dresden, 01062 Dresden, Germany

ABSTRACT

Microservice architectures became very popular in the last years. These small services can be developed, tested and deployed more easily than a single complex monolith. Another aspect which favors microservice architectures is their vast horizontal scalability under high load. But it is often overlooked if their performance is really superior compared to monolithic applications. We investigated test strategies on microservices and defined performance metrics in order to compare the performance of a web-based test application implemented as a monolith as well as a microservices re-implementation. The goal is to decide whether the effort was worth to re-implement our audience response system as microservices or not.

KEYWORDS

Microservices, Monolith, Performance, Tests, ARS

1. INTRODUCTION

The amount of software systems using microservice architectures is constantly increasing. With the concept of microservices, the separation of functional units is further driven to the point where each component becomes an independent service. This is mostly motivated by developer comfort. Smaller and simpler units can be developed, tested and deployed more easily compared to a single complex monolith. Different components of a service can be developed by separate teams. Each team can choose a development style, programming language and technology stack which fits their needs best. Bugs in one part of the application can be fixed faster by updating only the affected services and new features can be introduced smoothly without the need to change other code.

Another aspect which favors microservice architectures is their vast horizontal scalability. While a monolithic application can only be replicated as a whole and with possible side effects on consistency, microservices are a lot easier to scale. If they are properly designed, new instances of a service simply handle part of the requests made to this service. It is usually designed to handle many concurrent requests and operates better under heavy load than a single service. With small and independent services, the application's parts which experience the highest loads can be scaled separately from the lesser utilized parts. This reduces overall infrastructure costs compared to replicating the monolithic application. Maintainability and scalability were consistently ranked as the most important motivations for migration of monolithic applications to microservices [Taibi *et al.* 2017].

However, these advantages come at a cost: To ensure service interoperability their public interfaces need to be defined and documented prior to development. Additional tests for each service as well as for service-to-service communications are required. For the application as a whole, one needs to decide whether the frontend is aware of the different microservices and calls them separately or if it operates against a common backend. In the latter case, a new layer is introduced into the backend which is responsible for accepting and routing traffic to the different services. This is commonly called an *API gateway* and can additionally operate as a load balancer for the individual service instances. Furthermore, to handle complex requests that involve operations on multiple services, a communication overhead is introduced between the separate service

instances which doesn't exist in a monolithic application. Virtualization and containerization make the service deployment easier but introduce some overhead themselves.

In this paper we investigate performance testing strategies for microservices: First, we develop a test concept including different scenarios using single and multiple services. Then, we derive concrete test cases for our exemplary application under test – Auditorium Mobile Classroom Service (AMCS) – which is a web-based audience response system. Audience Response Systems (ARS) aim at enhancing the interactivity and quality of lectures by providing feedback to the students and the lecturer. They address problems like the lack of interactivity in huge university classes and facilitates learning in terms of an active, constructive and highly individual process. With AMCS the lecturer is able to construct learning questions, surveys and messages for distinctive students in the lecture [Kapp *et al.* 2014]. These interventions are delivered to the students during the session according to defined rules.

By using AMCS in huge university settings we recognized some performance lacks. Especially at the beginning of the lecture when all students login or register simultaneously or when answering the learning questions the performance breaks down. We made some further investigations to find the causes of this performance problems and identified the backend operations with multiple database interactions. That's why we decided to re-implement the backend as microservice application to separate the functionality into small services which can be scaled up when the system overloads.

We started with surveying open source load and performance testing tools. After detailed comparison of the features we chose one tool and implemented the different test scenarios. Then, we measured our application's performance using both the monolithic and the microservice backend. We compare and discuss the results in chapter 5. Our paper ends with a summary and outlook on further investigations.

2. CONCEPTS OF TESTING THE PERFORMANCE OF MICROSERVICES

We first explain the different scenarios how we tested our application. The overall goal is to load test as many facets as possible to create a full-blown picture of the application. This includes different configurations of nodes involved and their interconnection speed, different number of services and varying request loads.

a) Number of Services

The important thing to remember is that a microservice architecture appears as single interface to the end-user. However, it is possible that in one API request multiple services are involved in generating the response. Therefore, we categorized the test cases by number of services involved.

- *Single Service*: The single service scenario establishes the reference benchmark for response times. With one request we measure the round-trip-time from sending to receiving on client side. This is the minimal time needed for any request against the application under test.
- *Multiple Services*: To handle more complex requests, multiple services are involved into response generation. Thus we expect higher response times which include the duration of service-to-service communication. The performance then depends on the number of services, their corresponding IO operations and computing effort.

b) Varying Requests

To investigate the application's behaviour under different workloads it is necessary to vary the number of requests sent in a given period. We can distinguish between sending requests synchronously and concurrently which simulates the amount of users using the system under test in parallel.

c) Physical Distribution

We believe in a higher latency when all nodes are in different network domains. That is why we first test the services only deployed to a single host and after that they get deployed on multiple hosts, first all connected to the same network cluster and then to different clusters.

- *Single Host*: When deploying all services on one host the latency should be minimal because the services communicate via virtual networks. This is very fast.
- *Multiple Hosts*: In real applications the containers will be distributed over multiple machines to increase reliability and robustness. It makes a difference whether the containers are placed in one physical location or different ones.

- *Single Cluster*: Within the multiple host - single cluster scenario all services are distributed over multiple hosts which are placed in one physical location belonging to one data center. This means that all hosts have a high bandwidth connection, the resulting latency in service-to-service communication should be really low.
- *Multiple Clusters*: This scenario consists of multiple services on different hosts distributed over different cloud platform providers all over the world. It is highly scalable and reliable and not dependent on one single provider. This results in higher latency and additional effort to manage vendor specific infrastructure.

3. APPLYING TESTING CONCEPTS IN THE PROJECT

3.1 Architecture of the Test Applications

3.1.1 Frontend

We used the frontend of the monolithic application to test both backend implementations. It is a standalone JavaScript Single-Page-App (*AngularJS*) running on *nginx* web server communicating with the REST interface of the backend. It can easily be configured to work with both backends. Only the backend's address needed to be changed because our microservice implementation was build with the same public API as the pre-existing backend. Once the app was configured it could be packed and delivered by a separate *nginx* webserver which we ran inside our cluster.

3.1.2 Backend

a) Monolithic application: The monolithic application predates our microservice implementation and was used by us as it is. The backend is written in *Ruby* and uses a *Postgres* database. It serves all features to the clients via a REST API, because there also exist native apps for *Android* and *iOS*. All parts run as a single instance on the same virtual machine.

b) Microservice application: Our microservice implementation is written in Java and uses the Spring Cloud software stack. MongoDB document stores are used for persistence. The complete application architecture is shown in Figure 1.

Figure 1. Architecture of the AMCS microservice application

Our microservices are deployed on three virtual machines. Two of them are located on the same host system. To simplify the deployment we decided to fully containerize our application using *Docker*. Furthermore, we used Docker's own cluster management engine, the *swarm mode* [Dockers2019]. To share the service images across all nodes in our cluster we had to provide a custom docker registry as a service.

Spring Cloud comes with a great set of tools to implement a microservice application: It includes *Eureka*, a service registry which we used for routing and service discovery. The service-to-service communication is handled by *Feign* clients, which use *Ribbon* as a client-side load-balancer. We already ran a *nginx* webserver to provide our frontend. It seemed only natural to include the routing functionality for the backend requests in it. Additionally, the *nginx* configuration is very well documented and there were resources available explaining its use as an API gateway [Crilly2018].

3.2 Concrete Test Plans

We already introduced in chapter 2 the concept of different parameters we could test. The next step shown here implements a concrete test plan. In Table I each specific test is described by a scenario name, the involved services, the kind of database access and how the load in terms of how many users accessing the application is distributed over time. The different user load distributions we have chosen are described next:

- 1) 100 users try to execute the given test in parallel
- 2) 100 users get ramped up from 0 within 10 s and execute the given scenario
- 3) 10 users per second over 60 s start executing a test

Each of the tests runs in three configurations:

- The first configuration uses the monolithic application for the performance tests.
- The second one runs all microservices with their databases on two machines which are placed in the same computing center hosted by the same provider. That means low latency for all service-to-service and service-to-database calls.
- The third configuration includes placing the databases on a different machine than the microservices. The database machine is hosted by a different provider whose data center is located in another region.

Table 1. Concrete Test Setup

Test plan No.	Service scenario name	Involved services	Database access	Distribution of user load
01	Login with registration	Auth-S., User-S., Course-S., Lecture-S.	Read and write access	(1), (2), (3)
02	Login without registration	Auth-S., User-S., Course-S., Lecture-S.	Read access	(1), (2), (3)
03	Enroll into one course	Auth-S., User-S., Course-S., Lecture-S.	Read and write access	(1), (2), (3)
04	Get lectures	Lecture- Service	Read access	(1), (2), (3)

3.3 Expected Behaviour

This section is for pre-evaluation of the results before running the tests. Usually there is a gap between the imagination how the measurements will look like and how they are in reality. We expect a difference between tests which require only read access to the database and those which also need write access. For that reason test plan 01 should be slower than test plan 02. They are doing exactly the same excluding the injection of new users into the database. Test 04 will need a significant amount of time too because of its database writes. Test 04 should be the fastest. It does not write anything into database and performs no service-to-service communication.

- a) **Configuration dependency:** All tests which will run in the first configuration where the microservices and the databases are placed on two machines within the same computing center are expected to have a better overall performance. Database access needs more time than default inter-service communication. This leads to the expectation that a storage which is not in the same place than the services (configuration two) has longer response times. All in all every test runs in the second composition should take more time than its equivalent from the first one. In the microservice implementation there is an overhead because of service-to-service calls. The observation here should be that only test four takes a similar time compared to the monolithic application. In all other tests the monolith should be faster.
- b) **Load dependency:** Another important factor is load distribution. Within the first load scenario both architectures should have poor performance. They use the same reverse-proxy which has to handle all requests at once. Only after passing this bottleneck the microservices will be slower again because they are in a virtual docker network. This network is flooded by the incoming requests itself and additional by all service-to-service communication. The second and third load distribution scenarios are expected to perform nearly similar within the first ten seconds. After this scenario 02 stops requesting so the application can deal with the remaining execution where scenario 03 continuously fires up requests. We are not sure how both applications handle the permanent load but we hope that microservices will perform better here because the execution is distributed over the different services and must not be handled by one application.

4. IMPLEMENTING THE TEST CONCEPTS WITH GATLING

Before we implemented our concrete concepts, we had to decide for one load testing tool. We reviewed scientific sources [Leverger2017, Suominen2017] as well as practical guides [Lönn2017, Toledo2017] combined with our experience to find the best suited performance testing application. We decided to use *Gatling.io* for our performance test implementation. The decision is based on multiple factors. We wanted to use a new and modern tool. That said, *Apache JMeter* is not fulfilling our requirements. We then read some comparisons of performance testing tools in general and some only comparing *Gatling.io* and *Artillery.io*. *Gatling.io* was mentioned as more stable and adding less measurement error because of its implementation in *Scala* where *Artillery.io* could probably add more inaccuracy to the results. We also reviewed the available tutorials and official documentations. *Gatling.io* provided much more information and starting guides. The last point for our comparison was the extensibility of the provided DSL. Where *Gatling.io* tests can be completely written in the programming language *Scala*, *Artillery.io* tests can be provided by YAML configuration files or in JavaScript.

We started with the implementation of our tests by recording all sequences that should be tested. Recording here refers to just click through the website and also enter data, e.g. names or course pins, within a special browser mode which saves all input data into an HTTP-Archive (HAR). In the next step this archive was imported into *Gatling.io*. The tool then converted these data into its Domain Specific Language (DSL). The generated code needed a refactoring and some additions. We separated the single steps which we want to reuse, like login, logout or enroll. Furthermore, that separation enabled us to drive different test cases by combining the single steps in different orders and execute them in so called scenarios.

Because we want to test thousands of different users interacting with the application, a user feeder was implemented. This feeder dynamically insert data into a given template. The template here is the JSON string that is sent to login a user and the inserted data are numbered usernames and passwords. To authenticate requests the application uses *JSON Web Tokens*. These tokens are generated when a user gets logged in for the first time and is stored at the client side. With each request a token has to be sent otherwise the request will not be executed due to authorization errors. For our performance tests we needed to extract this token from the login response and store it for each user so that the next request of this user is executed with the correct token to ensure that not only one user executes one request multiple times.

Our tests should apply different user loads in order to observe potential bottlenecks. *Gatling.io* provides various workloads [Gatling2019]. We used the possibilities of executing a bunch of users at once, ramping up users distributed over a given duration and requesting with a constant amount of users per second for a given duration. Finally, we need to test our microservice application and also the monolithic reference application.

5. EVALUATION

In the following we will present the test results in the different test scenarios described in chapter 2.

5.1 Test Plan 1: Register and Login a New User

This is one of the challenging scenarios. It involves a service-to-service request when the *authentication service* queries the *user service* if the new user is already registered. Then, another POST request is issued to register the user, this involves a database write operation. Finally, a token is generated and returned.

1) Load distribution 1: 100 users at once: Sending all user login requests at once puts the server under a very high load. The monolithic application is able to respond to nearly all requests (94%) successfully with a mean response time of 2031 ms. We believe that this relatively high response time is caused by the sequential execution of database writes, forcing late-arriving requests to wait until the first requests are completed.

The microservice application shows different results: In both cases, with the database running in the same or an external cluster, all login requests fail. Investigating this failure further showed that all data inserts were performed in the database but its response time exceeded the timeout limitations set by the Spring components in their default configuration. In our opinion, the performance might be improved by experimenting with configuration settings and service replications. This is beyond the scope of this paper.

2) Load distribution 2: Increasing user count: In this scenario the number of concurrent user registrations is smaller than in the previous one because the number of users increases over a period of 10 s from 0 to 100. This means that some users are already logged in when new users try to register and the overall database load is less than with 100 users at once.

The monolithic application serves all requests successfully and within a mean response time of 126 ms. This is more than sixteen times faster than in the previous load distribution scenario. The microservice application on the other hand is still struggling with the database timeouts. But due to the smaller number of parallel requests, more return successfully: With the database running in the same cluster we observe that 8% of the requests succeed, with an externally running database 28% of the requests succeed. This implies that the higher latency of an external database is far less significant than the configuration restrictions.

3) Load distribution 3: Constant user load: As in the increasing load scenario, the monolith is able to process all requests quickly. Once again, the mean response time is very low at 111 ms.

With an internal database 2% of the requests sent to the microservice application return with success. The percentage improves to 74% when an external database is used. We believe the reason for this behavior is the overall limited computing power and memory consumption of the cluster. With the database instance running in the cluster the available resources are further diminished.

5.2 Test Plan 2: Login an Existing User

In this scenario the user is already registered and only a read operation is performed on the database. The credentials are compared and a token is generated and returned.

1) LD1: The monolithic application shows similar behavior as in the first test plan with a request success rate of 93% and a mean response time of 1276 ms. Database reads are faster than writes and can be performed in parallel explaining the overall faster response. This leads to the conclusion that the database is not the limiting part of the system which seems to be the application server. In the setting using a database inside the same cluster all requests are successful. The mean response time is 2837 ms. With an external database we achieved a 93% success rate and an equal response time. While here the microservice application needs more than twice the time of the monolith we think that in more advanced use cases or under higher loads replicating some of the services might improve the performance and ultimately outperform the monolithic design.

2) LD2: In this scenario all requests in both the monolithic and microservice application are successful. Differences arise once again in the response time: As expected, the monolith responds the fastest with 67 ms, the external database requires 805 ms and the internal database configuration is the slowest with 2673 ms.

3) LD3: Compared to the scenario which includes database writes the systems perform better in this one. Nearly all requests in each configuration were successful. Only the external database showed a 4% failure rate due to the timeouts. The mean response times differ vastly: The monolith only requires 68 ms. The microservice architecture needs 3223 ms with the external and 4535 ms with the internal database.

5.3 Test Plan 3: Enroll into a Course

In this scenario existing users are logged-in. Afterwards, each of them enrolls into the same course.

1) LD1: When this scenario is run against the monolithic application 5% of all requests fail. Whereas in both microservice configurations all calls are successful. The mean response time with the microservice application and an external database is 1819 ms and with the internal database 2124ms. This is significantly slower than the monolith with 912 ms. We conclude that the microservice application has a higher success rate because the overall load of this scenario is spread between different services. They can be run on different nodes in the cluster. Therefore, the individual workload for services and databases, e.g. the authentication service and user database for login and the course service and database for enrollment, is smaller compared to the monolith where a single application and database handles each request.

2) LD2: In this scenario the number of parallel users is constantly increasing but the average request rate is lower than before. Hence, all application configurations can handle it without failing. Once again the monolith is the fastest with a mean response time of 61 ms. The microservice system with an external database requires 789 ms and with an internal database 1360 ms.

3) **LD3:** This load distribution is very similar to the previous one in terms of the monolith's behavior. It handles all request successfully in a mean time of 81 ms. The microservice application is noticeably slower in both configurations (external database: 4251ms, internal database: 5460 ms).

5.4 Test Plan 4: Get Lectures

Finally we measured the performance of a single service request. We chose the "*Get lectures*" operation which only requires a single database lookup. Because this is such a simple request all systems are able to respond successfully under every user load scenario. We found that the time needed to create a response by the microservice architecture is independent from the database's location. In each scenario the monolith was about twice as fast as both microservice configurations. They still responded within 200 ms.

5.5 Manifestation of the Evaluation Results

In order to verify the previous results we decided to run some more test scenarios. We chose the third user load distribution with a constant amount of users per second because we found this to be very realistic in our use case. Furthermore, we increased the number of concurrent user requests to 20 users per second as it resembles the situation at the beginning of a lecture with a lot of students best. With the monolithic application we observed a higher number of long-running requests than before. 60% of requests took longer than 1200 ms, 23% of requests failed entirely. The mean response time of all successful requests was 1809 ms. The response time distribution can be seen in Figure 2.

Figure 2. Response time distribution of the monolithic application

Figure 3. Response time distribution of the microservice application

The number of active users increases because the application is unable to process the incoming requests in time. Backed by these results we tried to employ the advantages of the microservice architecture. We replicated two services for this scenario, the user and authentication service, to allow the application to handle the increased load. We found that with this configuration the microservice application performed better than the monolith. Only 14% of requests failed, 19% took longer than 1200 ms and the mean response time of the successful requests was 429 ms. The response time distribution can be seen in Figure 3.

After finishing and evaluating all test results there are some main takeaways: While we tested the microservice application with different database setups we found that the location of the database has no impact on the overall application performance. Therefore, the deployment of the database can be adjusted to the individual circumstances and capacities. One major drawback of the microservice architecture is the communication overhead between different services. It is clearly visible that the number of services involved in the processing of a single request is proportional to the delay of its response.

In all of the previously tested scenarios the monolith responded faster than the microservice architecture. As we expected there is a limit to the number of concurrent users the monolith is able to serve. If there are more user calls the monolith's performance breaks down. This is the point where more replicas in the microservice system are able to adapt better to the high load factor.

6. CONCLUSION AND FUTURE WORK

To conclude, both monolithic and microservice applications have strengths and weaknesses. We found that for small user loads of up to 10 users per second the monolith was the better choice because it had a higher success rate and shorter response times. If the user load increases the microservice architecture performs better because of its ability to distribute load between different replicas. Contrary to our starting hypothesis the database was not the bottleneck and its deployment does not have much of an influence on overall performance. As expected the inter-service communication overhead is the main problem of the microservice application.

Therefore, it is not always beneficial to choose a microservice approach which is also the outcome of the performance measurements done in [Ueda *et al.*2016]. If the goal is to refactor an existing monolithic application, detailed performance monitoring is required to find potential bottlenecks and attack surfaces for improvement. There won't be a performance gain simply by splitting the application into different services. For larger projects which see a lot of user traffic the microservice architecture can improve overall performance. The monolithic system is both physically and financially limited in its scalability, whereas the microservice application is able to replicate individual services as required to adapt to varying loads. This can be done dynamically as well as cost efficient. While microservices offer a lot of developer comfort by decoupling the individual teams a special focus needs to be put on API design. We found that having a single service for each domain class is not the preferable approach as it increases communication overhead. One should rather design the services following either segmentation in the internal business logic or external usage patterns. This requires more effort by the developer but improves performance.

From a technical perspective we hold the opinion that the Spring ecosystem is not ideal for developing microservice applications. It was indeed very easy to get started because it had all the batteries included for setting up a basic application structure. But once we decided to go deeper it became clear that the documentation is lacking and the configuration is ambiguous. Furthermore, choosing Spring binds you to the Java ecosystem for almost every part of your application. It adds components like load balancing and service discovery to the individual services without explicit declaration from the developer. In our mind this is contradictory to the microservice approach of technologically independent development teams. We prefer to keep the services as lean and "micro" as possible, simply reduced to the application logic they're encapsulating. Any configuration and infrastructure provisioning should be left to external tools. In our microservice implementation we started by using Spring's own components but switched to use external tools like docker, the swarm load balancer and nginx as an API gateway.

Finally, it is important to stress that our findings are solely valid for the examined ARS prototype. We do not want to condemn microservices in general. As this is a very new and constantly developing field further research in efficient components and their performance measurement as well as in optimization techniques is required [Heinrich *et al.*2017].

REFERENCES

- Crilly, L. 2018. *Deploying NGINX Plus as an API gateway, Part 1*. URL: <https://www.nginx.com/blog/deploying-nginx-plus-as-an-api-gateway-part-1/>.
- Docker Docs. 2019. *Swarm mode overview*. URL: <https://docs.docker.com/engine/swarm/>.
- Gatling.io. 2019. *Simulation setup*. URL: https://gatling.io/docs/3.0/general/simulation_setup/.
- Heinrich, R. et al., 2017. Performance Engineering for Microservices: Research Challenges and Directions. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. L'Aquila, Italy, pp. 223–226.
- Leverger, C., 2017. *Load testing fundamentals*. URL: http://colinleverger.fr/assets/projects/load_testing_fundamentals.pdf
- Lönn, R., 2017. *Open Source Load Testing Tool Review*. URL: <https://blog.loadimpact.com/open-source-load-testing-tool-review>.
- Kapp, F., Braun, I., Körndle, H., Schill, A., 2014. Metacognitive Support in University Lectures Provided via Mobile Devices. *Proceedings of the 6th International Conference on Computer Supported Education (CSEdu)*. Barcelona, Spain.
- Suominen, T., *Performance Testing REST APIs*. 2017. URL: https://www.theseus.fi/bitstream/handle/10024/140116/Suominen_Tero.pdf.
- Taibi, D., Lenarduzzi, V., Pahl, C., 2017. Processes, motivations, and issues for migrating to micro-services architectures: An empirical investigation. *Proceedings of IEEE Cloud Computing 4.5*, Honolulu, USA, pp. 22–32.
- Toledo, F. 2017. *Gatling vs. JMeter*. URL: <https://dzone.com/articles/gatling-vs-jmeter>.
- Ueda, T., Nakaike, T., Ohara, M., 2016. Workload characterization for microservices. *Proceedings of 2016 IEEE International Symposium on Workload Characterization (IISWC)*. Rhode Island, USA, pp. 1–10.