Database-Conscious End-to-End Testing for Reactive Systems using Containerization

Denton Wood[a] and Tomáš Černý[b]
Department of Computer Science, Baylor University, Waco, TX, U.S.A.

Keywords: Reactive Systems, Containerization, End-to-End Testing, Database, Docker.

Abstract: Reactive systems are a relatively new paradigm in computer science architecture with important implications for computer science. While much attention has been paid to effectively running end-to-end (E2E) tests on these architectures, little work has considered the implications of tests which modify the database. We propose a framework to group and orchestrate E2E tests based on data qualities across a series of parallel containerized application instances. The framework is designed to run completely independent tests in parallel while being mindful of system costs. We present a conceptual version of the framework and discuss future directions with this work.

1 INTRODUCTION

Reactive systems are quickly becoming a popular paradigm in software architecture. Version 2 of the Reactive Manifesto, a document detailing reactive system architecture for web applications, has over 26,000 signatures. The Manifesto defines a reactive system as a responsive, resilient, elastic, and message-driven system more suited for modern web application demands, including availability and data processing power (Bonér et al., 2014). Architectures such as microservices and serverless implement these tenets by breaking applications into scalable pieces which communicate with each other and which can be quickly deployed and redeployed on failure. Reactive system architecture promises resilient and responsive applications which will meet user demands.

In the past, performing extensive tests on reactive systems was difficult. In particular, performing end-to-end (E2E) tests, also called system tests (Amazon Web Services, 2017), was notoriously complex since all layers in an application must be booted to perform these tests. Reactive systems usually have many moving parts which testers needed to piece together to perform the tests. Additionally, performing the tests was time-consuming and often system-dependent. Containerization and container orchestration solutions such as Docker and Kubernetes have largely solved these problems by simplifying deployment, making it easier to boot applications for testing. Testers can then write automated tests in an external framework like Selenium and automatically run them using a continuous integration, continuous delivery (CI/CD) pipeline (Amazon Web Services, 2017) in frameworks such as Jenkins (Kawaguchi et al., 2021) or AWS CodePipeline (Amazon Web Services, 2019). Testers can even deploy tests in parallel using Selenium Grid (Selenium, 2021), decreasing the time cost of testing. This setup is so common that researchers have now begun to identify ways to streamline the process and make it more efficient and feasible (Augusto, 2020).

One area which has not undergone extensive research is E2E testing which modifies the database. Frameworks such as DBUnit (Gommeringer et al., 2012) are designed to test the state of the database; however, they do nothing to ensure that the test is repeatable later if the database is modified by the test. Some E2E tests add, modify, or remove database records; thus, tests must use a separate database designed for testing. Testers must reset this test database after each round of tests, and sometimes even between tests, in order to ensure that each round of tests is independent. Using only a single database for testing constrains tests which modify the database, or destructive E2E tests (Donahoo et al., 2021). Destructive E2E tests cannot be run in parallel with other destructive E2E tests or with nondestructive E2E tests (Donahoo et al., 2021) as this would violate test inde-

[a] https://orcid.org/0000-0003-1674-7451
[b] https://orcid.org/0000-0002-5882-5502

Wood, D. and Černý, T.
Database-Conscious End-to-End Testing for Reactive Systems using Containerization.
DOI: 10.5220/0010494403770383
In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 2*, pages 377-383
ISBN: 978-989-758-509-8

pendence and obstruct the ability to obtain repeatable test results. Thus, destructive E2E tests must be run serially, increasing the time cost of effectively testing a system.

While this seems challenging, we can apply the same frameworks and logic which enabled automated E2E testing in the first place to make E2E database-conscious testing feasible. Using a base database container image and certain customization SQL scripts for different tests, testers can effectively assign a database to each test to create a closed environment for testing. This would allow tests to be run in parallel, increasing utility and feasibility of E2E testing. To avoid imposing high resource costs on the testing infrastructure, we can group tests which do not modify the database or which require the resulting data from another test and run them serially on the same database.

In this paper, we propose a framework which accomplishes this and comment on our work in progress. Section 2 addresses related work to our field and comments on its limitations for our topic. Section 3 details our proposed framework. Section 4 details the work we have completed thus far, and Section 5 addresses our planned future work. Section 6 evaluates our framework's benefits and limitations. Finally, Section 7 closes the paper with some final remarks.

2 RELATED WORK

End-to-end testing of complex, interconnected systems is a known problem which multiple projects have attempted to solve. One of these is ElasTest (Bertolino et al., 2018), a distributed architecture designed explicitly for ”cloud testing.” It offers a language-independent framework for writing and deploying tests and uses container deployment to allow itself to stay application-agnostic. Since its inception, the ElasTest software (Gortázar et al., 2017) has continued to evolve and now supports features such as chaos testing, security testing, and external integrations such as Jenkins. However, ElasTest appears to lack the native ability to change the database between tests or specify a specific schema for a test. Other E2E testing frameworks which also lack this feature include AWS Device Farm (Amazon Web Services, 2021), an Amazon Web Services offering which only allows client-side application upload, and TestCraft (Testim, 2021), a codeless UI testing framework.

The high cost of running many E2E tests has led to research in limiting the resources used. Augusto (Augusto, 2020) addresses these concerns using a similar framework to our own, the RETORCH framework. It groups tests by resource usage, including levels of access (read, read/write, write-only, or dynamic) and additionally schedules them for maximum usage. While this framework is an excellent E2E testing tool, it does not address the problem of preparing databases for different tests or handling destructive E2E tests. Others propose limiting the number of tests in the regression test set, the set used to continually test the software (Ammann and Offutt, 2017). Gligoric et al. (Gligoric et al., 2015) present Ekstazi, a framework which automatically analyzes software modifications to select only the tests which would be affected by the change. Ekstazi has been shown to produce significant cost savings for production environments (Vasic et al., 2017). We believe that Ekstazi and similar efforts are complimentary to our work and that further investigation could lead to more cost savings for our framework.

More general efforts to test reactive systems vary. Schrammel et al. proposes the idea of test chaining, a way of grouping similar tests together which reduces execution time (Schrammel et al., 2013). This presents a great opportunity to reduce test data requirements by pairing these test chains with a single database; however, the authors make no mention of this strategy. Modern efforts focus on containerization. García et al. pairs ElasTest with Selenium WebDriver to construct an advanced user impersonation testing platform (Garcia et al., 2018). In a case study on implementing an E2E testing strategy, Lindell and Johnson create a Selenium testing apparatus using Docker containers and Microsoft Azure pipelines (Johnson and Lindell, 2020). However, neither of these address destructive E2E tests.

Research on destructive E2E testing is less common. Early works (Tsai et al., 2001) and (Xiaoying Bai et al., 2001) present E2E testing frameworks which include the ability to specify test database configurations. The introduction of containerization since that time makes this configuration much easier. The primary challenge of this project is ensuring that it is application-independent, decreasing coupling and enabling wider acceptance. Frameworks like ElasTest accomplish this, but at the cost of being able to directly manipulage the application database. We believe we can accomplish this by requiring a standard way of specifying the database address across all applications which will use the framework. The next section details our attempt at creating a suite to run database-swapping E2E tests on an existing reactive system.

3 APPROACH

Our proposed framework allows testers to run tests which modify the database while preserving independence between tests. We do this by effectively running these tests on completely separate databases using containers and images. To limit infrastructure costs, we group tests by the data they need in the database and by whether they will modify the database. Tests which do not modify the database can easily be scheduled by existing tools, so we primarily concern ourselves with tests which modify the database and how they interact with other tests.

3.1 Test Types

First, it is important to note the distinction between destructive and nondestructive E2E tests. Destructive E2E tests modify the database in some way and should not be run in conjunction with other tests. For example, creating a new user in an application would be a destructive E2E test. Nondestructive E2E tests are E2E tests which do not modify the database at all. For example, logging in to an application would be a nondestructive E2E test. Our argument specifically addresses the problem of running destructive E2E tests while maintaining test independence. Nondestructive E2E tests can easily be run on the same database with no consequence using existing tools.

Testing which includes a database must involve both destructive and nondestructive E2E testing to completely verify the integration. Nondestructive E2E testing ensures that the application can successfully connect to the database with the appropriate permissions required to access the tables it needs. It also ensures that the database has the appropriate schema for the application. Destructive E2E testing does all of the above, but it also checks that the application has permission to modify the database and that the database will receive and persist changes made by the application. While destructive E2E tests alone will cover all of the requirements of nondestructive E2E testing, utilizing both kinds of tests more specifically identifies issues with the database. For example, if the application cannot connect to the database, both kinds of tests will fail; however, if the application simply does not have permission to modify the database, only the destructive E2E tests will fail. In addition, certain application use cases simply do not modify the database, but should not be excluded from testing.

3.2 Testing System

To test the viability of database hot-swapping for E2E testing, we chose an existing system and constructed a testing suite for the system. The existing system used in the experiment is the MyICPC software (The International Collegiate Programming Contest, 2021), a web application developed for the International Collegiate Programming Contest. MyICPC is a social software which allows contest attendants to view the scoreboard for competing teams, access the contest schedule, see a Twitter feed of contest-related Tweets, and participate in a contest scavenger hunt. The application consists of a monolithic service and a series of microservices, all of which communicate with a cache backed by a database.

MyICPC meets all qualifications of a reactive system because it utilizes microservice architecture (MSA). It is responsive by ensuring a quick response to users through resilience and elasticity. MSA largely removes a single point of failure for the application; if one microservice fails, the rest of the system will continue to operate For example, if the database fails, the cache will continue to receive requests. MSA also allows elasticity through scalability; multiple copies of each microservice may be deployed to meet varying levels of system needs. Finally, MSA enforces a message-driven architecture since microservices communicate with each other using asynchronous techniques (ex. REST API calls).

We chose a particular microservice of MyICPC, the scoreboard service, to test. The scoreboard service is responsible for keeping a display of current scores for all teams competing in the competitions. It actively updates as teams solve problems and changes their score and ranking. Ideally, it would be testable by itself. However, we discovered that since the Twitter timeline is the first element that loads when users navigate to the MyICPC homepage, the timeline service must also be booted in order for E2E tests to work.

3.3 Conceptual Framework

Ideally, to have completely independent E2E tests, we would construct a database instance for each test. However, this approach does not remain feasible as the number of tests increases and imposes unnecessary production costs. Instead, we propose grouping tests which can run on the same database without interfering with each other. This decreases the number of independent databases required to run the tests.

In our proposed framework, each application would have a base database image built from the application DBMS' image (e.g. the MySQL or PostgreSQL image). The base image would supply the database schema and any sample data which is common across all tests. For each test, a tester could then specify a delta, or a series of commands to initialize data in a container built from the image. For RDBMSs, this would be a set of SQL statements. These commands would be run by the testing framework prior to the test execution. To allow testers to use the same delta across multiple tests, each delta specifier would be a reference to a file with the commands. The tester would also specify each test as nondestructive or destructive and any ordering between the tests (for example, test B can only be run after the database changes made by test A).

On startup, the testing framework would group tests in two ways:

- All nondestructive E2E tests with the same delta (or no delta)
- All nondestructive and destructive E2E tests with order specified and the same delta

Any test which cannot be grouped by the criteria above would be in a group of its own. We would not group tests together which have an ordering with each other but different deltas, as the second test would not see the changes from the first test. For this initial work, we do not consider tests which could modify different parts of the database at the same time, although this is an area for further research.

Figure 2 shows an example grouping of tests. All tests within Groups 1 and 2 are nondestructive and can be run in parallel with each other. This allows automatic in-group parallelization and possibly between-group parallelization if two testing databases are available at the same time. However, they cannot be run on the same database at the same time because they have different deltas. Group 3 consists of four tests which all have dependencies. The first destructive test makes modifications to the database which are required for the first nondestructive test and the second destructive test, establishing a dependency. The second destructive test makes changes to the database which are required for the second nondestructive test, creating an additional dependency. This establishes a serial schedule for the group. Group 4 consists of a lone test which was broken off from Group 3 as an optimization to ensure no one test group is too large. If the test environment has enough database containers available, the two groups can be run in parallel; otherwise, they will execute serially.

The testing framework would then boot a user-specified number of instances of the application. It would additionally boot a larger number of database

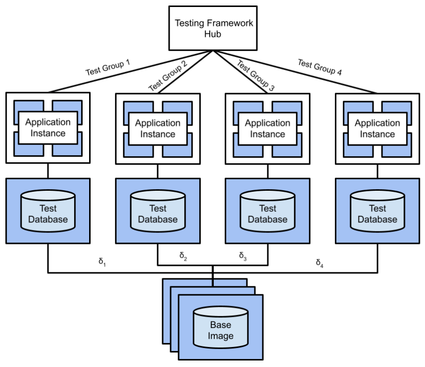

Figure 1: Diagram of Proposed Framework.

instances: one for each of the application instances and a warm pool of additional instances. Each instance would be a container based on the base database image for the application. The framework would then assign each group of tests to an instance of the application, initialize the instance's database with the delta, and run the tests. Upon completion, the testing framework would swap the current instance of the database with a warm instance, destroy the used container, and create a new container from the application's base database image. This architecture is based on Amazon Web Services' Provisioned Concurrency mode for Lambda instances (Beswick, 2019). Large groups of tests would be broken up across multiple application instances to increase parallelism. See Figure 1 for a diagram of the proposed framework.

4 CURRENT WORK

To make MyICPC suitable for our experiments, we have containerized all services which were not already containerized by the original developers by constructing Docker (Merkel et al., 2014) image files. We tested two orchestration tools for the containers: Docker Compose (Docker Inc., 2019) and Kubernetes (The Kubernetes Authors, 2021). While Docker Compose is more tightly integrated with Docker, we chose Kubernetes for its automatic scaling capabilities. We then created a Kubernetes kustomization configuration to boot the application in a scalable manner.

We have additionally constructed an initial deployment of the testing framework which runs tests on a pre-deployed instance of MyICPC. The framework uses Selenium (Huggins et al., 2021), a user interface testing tool, to manipulate the MyICPC application and simulate user actions. The tests themselves are written in JUnit (JUnit, 2021), a popular testing framework for Java, using the Selenium WebDriver

API. We parallelized the tests by running headless Chrome and Firefox browser nodes in Docker containers which are connected to a Selenium Grid v3 hub, also in Docker. This allows us to keep browser versions constant no matter where the tests are run and avoid test brittleness problems common with user interface testing.

5 EVALUATION

In this section, we evaluate our proposed framework's benefits and drawbacks.

5.1 Benefits

We believe our work is a novel approach to E2E testing which makes it much easier to set up and execute. Tests must be repeatable (Ammann and Offutt, 2017), which has traditionally made automating destructive E2E testing difficult due to the cleanup required afterward. Our approach makes no persistent modifications to any databases, ensuring that the tests will always be repeatable. Since the databases are entirely containerized with no volumes, they spin up and spin down with no infrastructure left behind. This allows multiple testers to test at once on the same server, provided that each has a different installation on which to test. This could allow the approach to be constructed into a full-fledged provided service similar to AWS Device Farm (Amazon Web Services, 2021) and injected into a CI/CD pipeline. Because it is built using popular containerization softwares Docker and Kubernetes, it can be run across multiple servers and operating systems automatically and scaled as far as the Kubernetes cluster will allow.

Our approach additionally enables parallelization of destructive E2E tests by executing them on simultaneously running test databases. This increases the number of destructive E2E tests that testers can perform in a short amount of time, which is an important barrier to overcome when considering implementing E2E testing. We believe this would allow developers to begin writing E2E tests for more specific use cases, increasing E2E coverage and bug discovery for a better overall user experience.

5.2 Limitations

Our work is primarily limited by the available infrastructure the user has for testing. Constructing multiple databases may be too expensive for some software teams, especially if the test data required is quite large. Base database images and deltas must be very small in order to run the framework effectively, which may reduce the scope of E2E tests which can leverage the framework. Scalability is additionally limited by the number of servers which are connected to Kubernetes.

Additionally, our framework would require the application to be completely containerized and orchestrated by a configuration file (i.e. a Kubernetes ”kustomization” file) for the tests to run. Otherwise, it would not be able to spin up multiple instances of the application as it would have no base image from which to do so. While containerization has become quite popular for deploying applications, many legacy applications would not be able to take advantage of our framework.

Finally, this is a very conceptual framework. There are many opportunities and obstacles which we have not discovered yet and are not addressed here. We hope to obtain positive results from our implementation as discussed below.

6 FUTURE WORK

This paper presents the conceptual framework of the project. Moving forward, we will begin implementation and experimentation. Our goals for the project are as follows:

- Easy to implement. The project should be able to be quickly installed, and configuration should not be required to be complex.
- Integratable. Along with point 1, the project should be integrated with popular build and test frameworks. For Java, this would include Maven and JUnit.
- Intelligent. The framework should parallelize where possible to decrease time to completion.

Initially, the project implementation will be split into three parts: database integration, test grouping, and scalability.

To implement the database hot-swapping functionality, we would leverage Kubernetes to spin up a series of database instances. We would then issue kill commands to drop containers once testing is completed and reassign the existing application instance to a waiting database instance. This would prevent us from having to continually kill and create instances of the application itself.

To group the tests, we will specify an API which users can import which uses annotations. As specified above, users will denote their tests which are destructive and specify any database deltas required to run

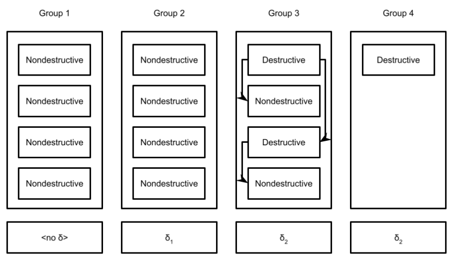

Figure 2: Diagram of Proposed Framework.

them. An example test in JUnit might look like the following:

```
@Destructive
@Delta("add-test-user.sql")
@Test
public void addPointsToTeam() {
   ...
}
```

We would then write a startup script to detect the annotations and group the tests accordingly. See Figure 2 for a sample grouping. As the project expands, this could also include grouping tests which modify different parts of the database at the same time. However, for the initial implementation, we would enforce a single-writer model over the entire database.

To allow the application to be tested at scale, we would require users to supply a Kubernetes or Docker Compose configuration of their application with an environment variable (ex. $DATABASE_HOST) set in their applications to access the database. We could then dynamically set the database host at runtime to supply the correct database. Additionally, we plan to offer the ability to scale the testing framework across multiple machines using Kubernetes, allowing testers to scale up their infrastructure as needed.

We have presented a possible framework for running E2E tests with database hot-swapping on an application using containerization. We aim to continue this work and demonstrate the feasibility of controlling the database during E2E testing. We hope to promote implementation of E2E testing with multiple database scenarios.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research.

REFERENCES

Amazon Web Services (2017). Practicing Continuous Integration and Continuous Delivery on AWS.

Amazon Web Services (2019). Modern Application Development on AWS. page 41.

Amazon Web Services (2021). Overview of Amazon Web Services - AWS Whitepaper.

Ammann, P. and Offutt, J. (2017). *Introduction to Software Testing*. Cambridge University Press, second edition.

Augusto, C. (2020). Efficient test execution in end to end testing: resource optimization in end to end testing through a smart resource characterization and orchestration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, pages 152–154, New York, NY, USA. Association for Computing Machinery.

Bertolino, A., Calabró, A., Angelis, G. D., Gallego, M., García, B., and Gortázar, F. (2018). When the Testing Gets Tough, the Tough Get ElasTest. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 17–20. ISSN: 2574-1934.

Beswick, J. (2019). New for AWS Lambda – Predictable start-up times with Provisioned Concurrency. https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency.

Bonér, J., Farley, D., Kuhn, R., and Thompson, M. (2014). The Reactive Manifesto. https://www.reactivemanifesto.org.

Docker Inc. (2019). Compose file version 3 reference. https://docs.docker.com/compose/compose-file.

Donahoo, M. J. et al. (2021). ICPC Developer Documentation. https://icpc.global.

Garcia, B., Gallego, M., Santos, C., Jimenez, E., Leal, K., and Fernanez, L. (2018). Extending WebDriver: A Cloud Approach. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 143–146, Coimbra. IEEE.

Gligoric, M., Eloussi, L., and Marinov, D. (2015). Ekstazi: lightweight test selection. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 713–716, Florence, Italy. IEEE Press.

Gommeringer, M. et al. (2012). DbUnit. http://www.dbunit.org.

Gortázar, F., Gallego, M., García, B., Carella, G. A., Pauls, M., and Gheorghe-Pop, I. (2017). Elastest — an open source project for testing distributed applications with failure injection. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–2.

Huggins, J. et al. (2021). Selenium - Web Browser Automation. https://docs.seleniumhq.org.

Johnson, T. and Lindell, C. (2020). *Docker Image Selenium Test : A proof of concept for automating testing*.

JUnit (2021). JUnit 5. https://junit.org/junit5/.

Kawaguchi, K. et al. (2021). Jenkins. https://www.jenkins.io/index.html.

Merkel, D. et al. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239).

Schrammel, P., Melham, T., and Kroening, D. (2013). Chaining Test Cases for Reactive System Testing. In Yenigün, H., Yilmaz, C., and Ulrich, A., editors, *Testing Software and Systems*, Lecture Notes in Computer Science, pages 133–148. Springer Berlin Heidelberg.

Selenium (2021). Grid :: Documentation for Selenium. https://www.selenium.dev/documentation/en/grid.

Testim (2021). TestCraft - Codeless Test Automation. https://www.testcraft.io.

The International Collegiate Programming Contest (2021). MyICPC. https://my.icpc.global.

The Kubernetes Authors (2021). Kubernetes. https://kubernetes.io.

Tsai, W. T., Xiaoying Bai, Paul, R., Weiguang Shao, and Agarwal, V. (2001). End-to-end integration testing design. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 166–171. ISSN: 0730-3157.

Vasic, M., Parvez, Z., Milicevic, A., and Gligoric, M. (2017). File-level vs. module-level regression test selection for .NET. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 848–853, New York, NY, USA. Association for Computing Machinery.

Xiaoying Bai, Tsai, W. T., Paul, R., Techeng Shen, and Bing Li (2001). Distributed end-to-end testing management. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, pages 140–151.