UDC: 004.774:004.415.53
Professional paper

TESTING STRATEGIES IN MVC-BASED APPLICATIONS: MODEL, VIEW, AND CONTROLLER STRATEGIES

Elira ABDURAMANI, Agon MEMETI [0000-0002-6824-3856], Florim IDRIZI [0000-0001-7514-3282], Shkurte LUMA-OSMANI [0000-0003-2464-249X]

Department of Computer Sciences, Faculty of Natural Sciences and Mathematics, University of Tetova, Tetova, North Macedonia
e.abduramani320043@unite.edu.mk

Abstract

Much like other modern technologies, automated testing is rapidly advancing in today's fast-changing software landscape. Keeping up with and understanding the numerous emerging tools become essential. This review offers a critical evaluation of three automated modern frameworks designed for unit testing MVC architectures. Most tools cover only one layer of the MVC - Model, View and Controller - so reliable unit tests are a challenge. Three complementary solutions that together promise full-stack coverage are reviewed here. With ModelWeb, simple flowchart model user interactions are converted to run BDD-Selenium tests with maintainable scenarios. WebExplor is an AI-powered curiosity agent that navigates dynamic UIs to reveal small, hard-to-find bugs. Finally, we review ten top REST API testing tools and assess their performance in driving real world services, showing strengths in automation and weaknesses in call sequencing and input generation. Comparing each approach's methodology, coverage, and maintainability, we find that no tool can stand alone. Mixing model-based clarity, intelligent exploration, and stringent API validation instead provides the strongest and most adaptive path towards end-to-end MVC testing.

Keywords: MVC Architecture, Web Application Testing, Test Automation Tools, AI-Driven Testing.

1 Introduction

The architectural pattern made out of Model, View and Controller (MVC) powers a vast number of frameworks like ASP.NET MVC, Spring MVC and Angular by separating concerns in this order data (Model), UI(View) and logic (Controller). Automated unit tests in CI/CD pipelines are essential for catching regressions as soon as possible but manual test writing can be very time-consuming and not very reliable while considering that UIs are evolving in a record time. Many tools used for testing focus only on one of the layers-models, views or controllers in which case they leave coverage gaps. This critical review asks the question: How can teams achieve broad, maintainable unit tests across all the MVC layers without needing excessive effort? We examine and analyze three different frameworks:

- **ModelWeb:** Converts flowcharts into BDD scripts covering models, view and controllers.
- **WebExplor**: Uses reinforcement learning to discover workflows and state transitions.
- **RESTful Tools**: Performs exhaustive AOI-driven validation of controller and model logic.

By analyzing how each tool generates tests, what parts of the application it covers, how well the aforementioned tool fits into CI/CD pipelines and how it handles changes we find that each of them have unique strengths. Later on, we suggest a combined approach that uses flowcharts for clarity and understanding, smart UI exploration and strong API testing to fully cover the MVC applications.

2 Related Work

ModelWeb, was introduced as a model driven testing toolset that empowers testers to specify web application behavior through flowchart models [1]. A tester uses a graphical editor to model user actions and these visual models formally capture the intended flow of interactions for each functionality. ModelWeb then proceeds to translate each flowchart into BDD (behavior driven development) scenarios expressed in the Gherkin syntax (Given-When-Then) [1]. Consequently ModelWeb reduced the time needed to create texts by 27-41% compared to traditional manual scripting, at the same time, it enabled users to generate 51-113% more test scenarios. It also maximizes its testing throughput while minimizing required skill sets. Despite its undeniable advantages, ModelWeb possesses some certain weaknesses common to model-based testing approaches. The framework's effectiveness hinges on the quality of the user-defined flowchart models. If the testers model does not include certain pages or paths, those paths or parts of the app will not be tested.
Modern web apps often update their content dynamically (e.g with JavaScript), which static flowcharts in ModelWeb struggle to represent. As the app changes, testers must update the flowcharts and tests which can result to be really time-consuming, especially for large systems. ModelWeb has only been tested on small demo apps in controlled settings. While early results have been promising, it's performance in a large, real-world MVC systems and CI/CD pipelines still needs to be proven. When compared to ModelWeb's manual modeling, WebExplor [2] presents a total opposite automatic web testing framework which uses curiosity-driven reinforcement learning (RL) to autonomously explore web applications. It treats the application as an environment in which an agent navigates through the user interface (UI), very similar to a human tester but guided by an intelligent algorithm. This approach defines states (based on the web page DOMs) and actions (clicking links, filling forms) for the RL agent and uses a curiosity reward to encourage exploration of unseen states. WebExplor incrementally builds an internal model of the web application's navigation structure during the exploration. This learned model serves as a high-level guide, tracking the visited states or pages to reduce redundant actions and prioritize the interactions that are yet unexplored. Anyway, in evaluations of sic real-world open-source web projects and a commercial SaaS application, WebExplor it found more bugs, covered more code and tested more efficiently compared to state-of-the-art methods. The RL-driven approach proved to be especially good at discovering complex bugs that normally require long sequences of action and specific inputs-often missed by random testing. WebExplor found 12 previously unknown failures in a production web app, all of which were later confirmed and fixed by the developers [2]. It is currently pushing the frontier of AI-driven testing, it also raises challenges of applying RL in the web domain. One of the problems is state explosion problem. In order to change the focus from testing the user interface (GUI) to testing the server part of modern web applications, authors did an empirical study of REST API testing tools [3] and reviewed automated methods for testing REST APIs, which act as the Controller in an MVC setup.

Knowing that the backend handles key logic and data, testing the same is essential and rather it should be done with the utmost care and focus to make sure that it works as well as expected. Instead of proposing a new tool, these authors critically examine 10 state-of-the-art API test generation tools, taken from academia and industry to evaluate how the current testing techniques can automatically exercise REST APIs [3]. The results of this study ultimately reveal that there is room for improvement across the board because even the most modern automated tools achieve low coverage on many APIs. On average, every tool missed a significant portion of the API logic, no tool was able to exercise most of the code. One of the most primary problems was the difficulty of generating valid and diverse inputs for API endpoints. Many tools struggle with inputs that must satisfy specific formats. Another important issue or a limitation we might call it is handling dependencies between API calls. Real-world services often require calling endpoints to a certain order, for example one must POST an object before GETing it. The study found that most tools either ignore such stateful sequences or rely on simplistic heuristics which leads to many failed and ineffective calls [3]. While this study focuses on effective approaches for end-to-end MVC testing, similar research on learning system optimization [4-5] demonstrates that strategic use of MVC architecture with Blazor can significantly improve the efficiency and responsiveness of complex systems.

3 Comparative Evaluation of Three MVC Testing Strategies

All three papers have the objective of automating web application testing however they diverge significantly in methodology and scope. ModelWeb and WebExplor both focus more on web GUI (front-end) testing, but at the same time they represent opposite ends of the spectrum. One is guided by human-designed models whereas the other one by machine learning and exploration. Another key difference lies in how mature and realistic the evaluations are. WebExplor and the API testing tools were tested in real-world systems, including a commercial SaaS app and widely used open-source projects. This shows that they can work in production-like environments. On the other hand, ModelWeb has only evaluated and tested on small prototypes which had a small number of users, so it's unknown how well it will work and adapt in large
enterprise applications. Together, these three studies mark significant progress in automating the testing of MVC web applications, especially in areas that typically demand a lot of human effort. The detailed comparison is presented in table 1.

Table 1. Comparison of Three MVC Testing Strategies

Framework	Model Layer	View Layer	Controller Layer	Notes
ModelWeb	✓	✓	✓	Flowchart → BDD tests; requires manual models
WebExplor	-	✓	✓	RL-driven; UI exploration; no upfront models
APItools	✓	-	✓	Automated REST calls; input sequencing issues

3.1 Implementation of Unit and Integration Testing: This part describes the unit testing and integration testing of the StudentApplication project. The testing strategy involves creating unit tests in the interests of guaranteeing service logic, and controller tests for guaranteeing proper API endpoint functionality. The overall design of the test project is into folders and files that well define unit testing responsibility and integration testing responsibility. The Tests folder contains test classes like UnitTest1.cs and StudentControllerTest.cs. The testing framework employed throughout the project is NUnit, which offers a good and flexible base on which to write and run tests.

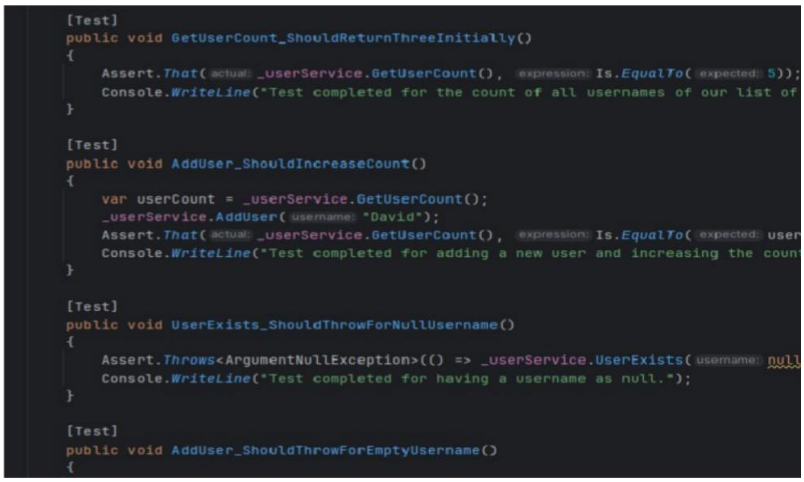

Figure 1. UniteTesting Code

These are unit tests, thus do not rely on the database, HTTP context, or any other external systems. One can now test reproducibly and quickly the inner logic. For simplifying test classes and not repeating using statements, a shared using file called GlobalUsing.cs was included. This import makes all test classes automatically aware of the NUnit framework without importing it explicitly in each file. This approach renders the code cleaner and complies with best practices in large-scale test projects.

The UserService class in the main application has methods dealing with simple operations like user creation, validation, or business rules. It is the class under test in the UnitTest1.cs. The test method in the UnitTest1.cs indeed tests a method of UserService to check whether the business logic layer of the application is appropriate.

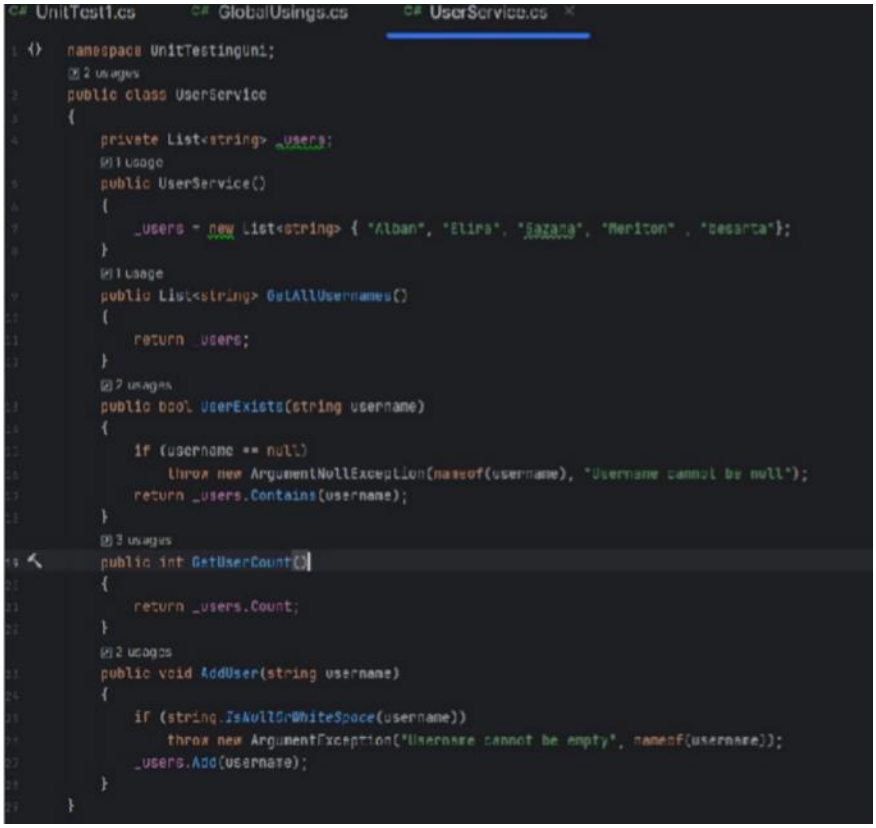

Figure 2. UserService Code

The test method in the UnitTest1.cs indeed tests a method of UserService to check whether the business logic layer of the application is appropriate.

3.2 Successful Execution of Unit Tests

After performing the test steps, the unit tests were run by using Visual Studio's Test Explorer. As depicted below, the 9 unit tests were run successfully, indicating the service logic is properly executed in all scenarios that are being tested. For verification that there is complete information about the outcome, another screen shot of the whole list of all 9 individual test cases with every test case marked as "Passed" was taken. This indicates that UserService performed as anticipated in all the test cases.

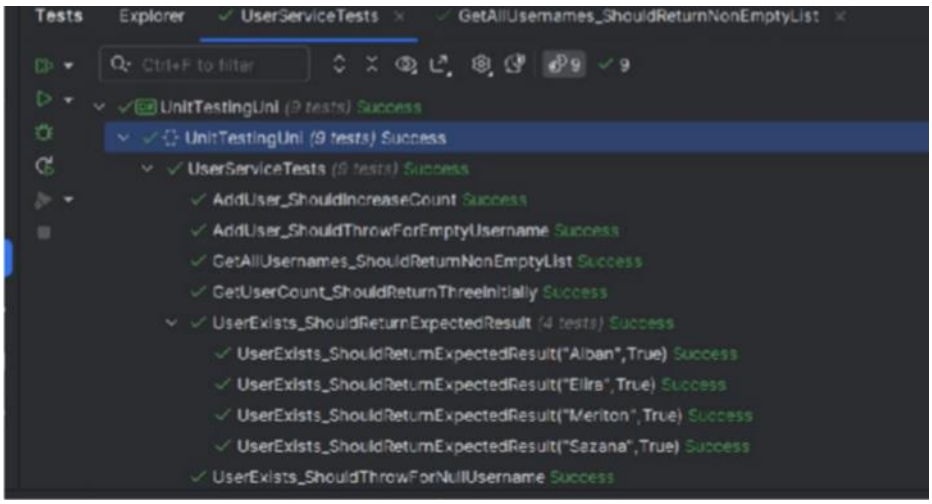

Figure 3. All Nine Passed Test Cases in Test Explorer

This efficient deployment ensures that business logic within UserService fulfills anticipated functional expectations. Testing at this level offers considerable assurance that the application's service layer is robust and dependable.

3.3 StudentApplication Test Structure: Apart from unit testing, integration testing was also conducted on the API controller level. A different hierarchy was followed especially for testing StudentApplication project. These types of files are studentcontroller test.cs, which is specifically meant for endpoint validation.

3.4 StudentControllerTest.cs – Integration Testing: The StudentControllerTest.cs file holds tests that mimic HTTP requests to the controller level and validate API response behavior. They check for endpoint functionality including:

- Returning suitable status codes
- Maintaining data consistency

Figure 4. Code Implementation of Studentcontrollertest.Cs

3.5 Execution and Results of Integration Tests: To run the integration tests, the following command was executed via terminal: dotnet test StudentApplication.Test. Terminal output displayed confirmation of the pass of 7 integration tests, displayed in green, including a quick report on passed tests, run time, and general status.

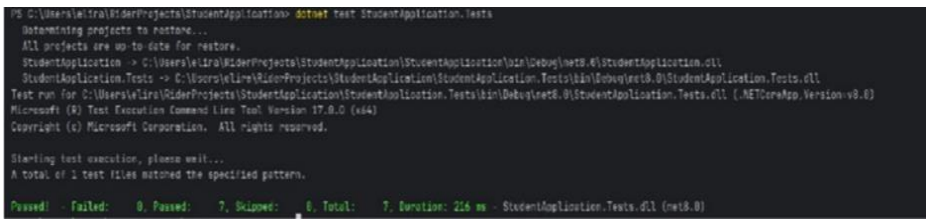

Figure 5. Passed Seven Integration Tests

Terminal output displayed confirmation of the pass of 7 integration tests, displayed in green, including a quick report on passed tests, run time, and general status.

The implementation showed a thorough testing approach that included integration tests for API endpoints and unit tests for the main service logic. Every test was completed successfully, verifying the accuracy of the system's behavior as well as the isolated components. A strong

and dependable basis for upcoming development and deployment is guaranteed by this implementation.

4 Conclusion

Effective MVC testing involves several complementary techniques. ModelWeb converts flowcharts to easy to maintain BDD tests covering defined user flows; WebExplor learns to probe UIs for hidden bugs based on reinforcement learning. And automated REST API tools validate back-end logic. None of the approaches alone is sufficient but taken together they cover the whole MVC stack. Uniting model-based clarity, AI-driven exploration, and API-level rigor, teams can create maintainable test suites that scale with applications. For now, research should be focused on seamless frameworks that combine these methods, self-healing tests that evolve with UI and API changes and advanced AI techniques like deep learning for UI understanding and intelligent fuzzing for ever more complex scenarios.

References

[1] **Ozkaya, M., Kose, M.A., Mamur, A.B., Koc, T.** (2022). ModelWeb: a toolset for the model-based testing of web applications. *Annals of Computer Science and Information Systems*, **32**, 331–338.

[2] **Y.Z. et al.** (2021). Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In: *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 423–435. IEEE. https://doi.org/10.1109/icse43902.2021.00048

[3] **Kim, Q.X.S.S., and O.M.A.** (2022). Automated test generation for REST APIs: no time to rest yet. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, July 2022. ACM.

[4] **Memeti, A., Iljazi, A., & Luma-Osmani, S.** (2024). Optimizing Educational Data Storage and Retrieval in Learning Management Systems Using Table-Valued Parameters with Blazor Components. In *Proceedings of the Third International Conference on Innovations in Computing Research (ICR '24)* (pp. 390–398). Springer Nature. https://doi.org/10.1007/978-3-031-65522-7_35

[5] **Huseini, K., Memeti, A., Imeri, F., & Luma-Osmani, S.** (2025). Optimizing Learning Experiences: MVC BLAZOR Components and Emerging Educational Technologies in LMS Development. In *Proceedings of the Fourth International Conference on Innovations in Computing Research (ICR'25)* (pp. 331–345). Springer Nature Switzerland AG. https://doi.org/10.1007/978-3-031-95652-2_28