Global Journal of Computer Science and Technology: E
Network, Web & Security
Volume 16 Issue 7 Version 1.0 Year 2016
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Automock: Automated Mock Backend Generation for Javascript based Applications

By Neha Singhal & Harshit Jain

Abstract- Modern web development is an intensely collaborative process. Frontend Developers, Backend Developers and Quality Assurance Engineers are integral cogs of a development machine. Frontend developers constantly juggle developing new features, fixing bugs and writing good unit test cases. Achieving this is sometimes difficult as frontend developers are not able to utilize their time completely. They have to wait for the backend to be ready and wait for pages to load during iterations. This paper proposes an approach that enables frontend developers to quickly generate a mock backend that behaves exactly like their actual backend. This generated mock backend minimizes the dependency between frontend developers and backend developers, since both the teams can now utilize the entire sprint duration efficiently.

Keywords: javascript development; xml http request; javascript testing; web development; automated mock server.

GJCST-E Classification: D.1.1

AUTOMOCKAUTOMATEDMOCKBACKENDGENERATIONFORJAVASCRIPTBASEDAPPLICATIONS

Automock: Automated Mock Backend Generation for Javascript based Applications

Neha Singhal [α] & Harshit Jain [σ]

Abstract- Modern web development is an intensely collaborative process. Frontend Developers, Backend Developers and Quality Assurance Engineers are integral cogs of a development machine. Frontend developers constantly juggle developing new features, fixing bugs and writing good unit test cases. Achieving this is sometimes difficult as frontend developers are not able to utilize their time completely. They have to wait for the backend to be ready and wait for pages to load during iterations.

This paper proposes an approach that enables frontend developers to quickly generate a mock backend that behaves exactly like their actual backend. This generated mock backend minimizes the dependency between frontend developers and backend developers, since both the teams can now utilize the entire sprint duration efficiently. The approach also aids the frontend developer to perform quicker iterations and modifications to his or her code.

Keywords: javascript development; xml http request; javascript testing; web development; automated mock server.

I. Introduction

The modern development process is increasingly moving towards an Agile Workflow. It is a process followed by teams both large and small. There has been a paradigm shift from long, slow development cycles to quick iterations. Agile processes have also been documented in multiple research papers [4; 5; 9].

The Agile approach is also followed for web application development (including development of Single Page Applications). A modern web application generally comprises two integral components—the frontend (or the UI) and the backend server. Both run in tandem and are heavily dependent on each other. The frontend depends on the backend for data and the backend relies on the frontend to display the content to the end user.

A typical development sprint is comprised of three major phases. First is the assignment of features to the frontend team and the corresponding backend team. Post the assignment phase, the sprint moves to the feature implementation stage. At this stage, Backend developers work on implementing the server features. The frontend developers have to generally wait for the backend to be ready. Once the backend is ready, the frontend developers implement the user interface. The backend developers are mostly idle during this time. One of the major challenges faced during development is that the non-production environments of integrated third-party services are unstable and not accessible at times, blocking developers from interacting with these services.

The final stage is the User interface (UI) unit testing stage. Post feature implementation, the developer has to write test cases for his or her module. There are some frequent issues usually faced at this point. Firstly, UI test cases for asynchronous network calls are messy and time consuming to write. Secondly, UI test cases that make network calls consume a lot of time in execution. Thirdly, UI test cases generally require consistent data based on real-world data. Finally, UI test cases must not add any test data to the database.

II. Proposed Model

Our approach resolves some of these issues faced by frontend developers. It has an intuitive interface and can easily be integrated into most JavaScript based applications with a single line of code.

The key features of our approach are:

- A fully-functional mock server
- Very lightweight; comprises just a single JavaScript file
- Flexibility to support as many API calls as required
- Automatic capture of any existing API calls and generation of mock data for them
- Integration into existing applications with a single line of code
- Support for polymorphic responses:
  - o Alternate error responses for an API call
  - o Multiple configuration based responses for the same API call
- No interaction with database

*Author α σ: Adobe Systems Incorporated, Bengaluru, Karnataka, India.
e-mails: nsinghal@adobe.com, hajain@adobe.com*

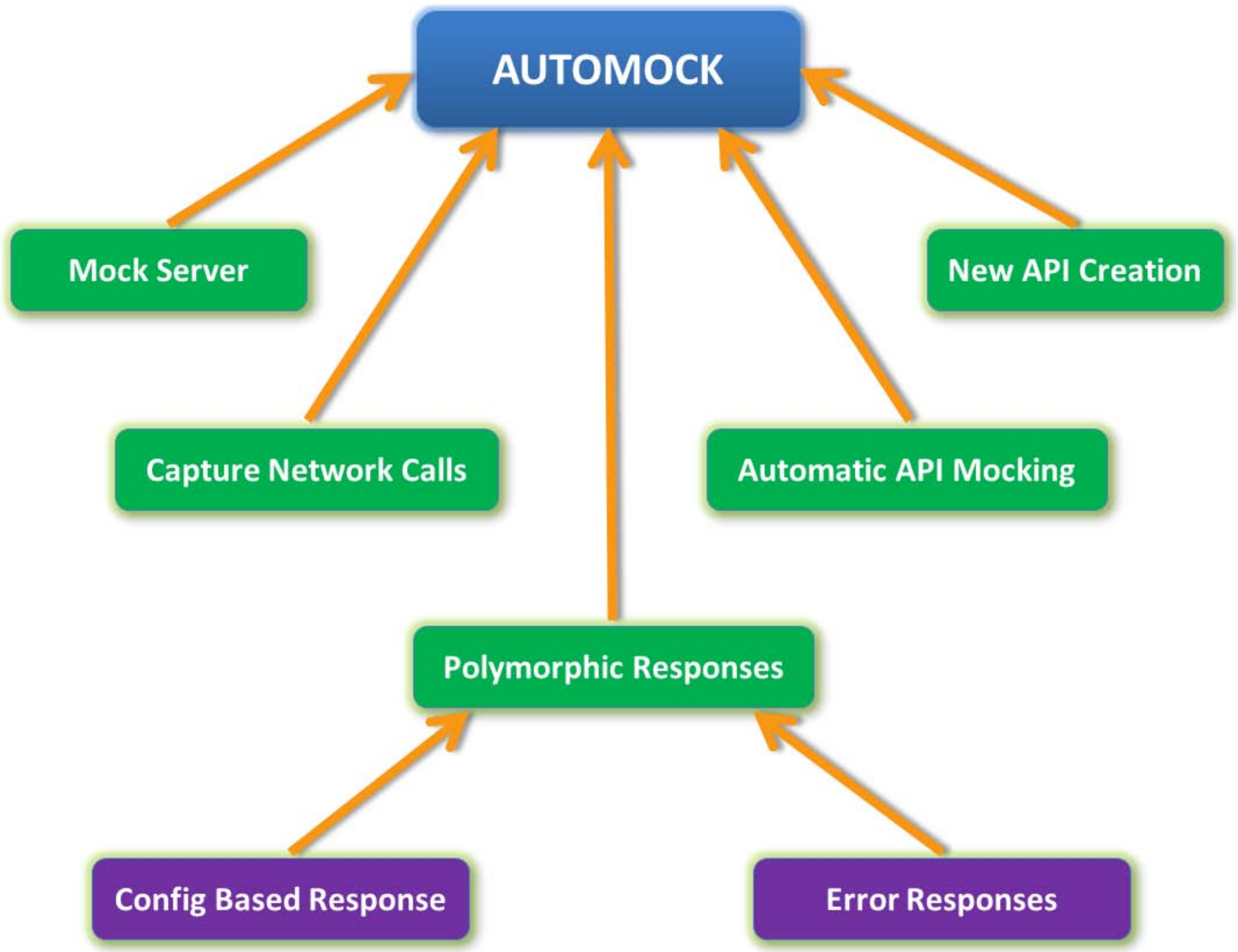

Figure 1: Overview of Automock features

Our approach is best suited for any medium to large-sized JavaScript web application including applications working with third-party components. It also designed for JavaScript unit testing. It is especially suited for interdependent teams working on the same web application in parallel.

As of now, the only limitation with our approach is that it only supports web development projects which use JavaScript.

Detailed description of our approach:

a) Fully functional mock server

A backend server comprises of a mapping between API calls and the corresponding responses for those calls. The frontend of a web application usually uses the AJAX (Asynchronous JavaScript and XML) protocol [6] to query the backend server. Though this allows the application to provide a user with a rich and interactive user experience, it also imposes certain challenges. The XML Http Request Spec [15] on which AJAX is based is browser implemented and hard for an application to control directly. To make a network call, the JavaScript code in the application calls the XML Http Request Object of the browser the application is running in directly. The interaction between the application and the XML Http Request Object is done through a series of callbacks. Once the network call is made, the server returns the appropriate response to the caller (based on the API request made). The browser then passes this information along to the application (through the aforementioned callback).

The XML Http Request Object according to the specification is meant to be immutable. Applications are not allowed to edit it directly without also manually implementing the overridden functionality. Our approach achieves the same functionality as a normal XML Http Request Object without the application realizing that the XML Http Request Object is being intercepted. Our approach achieves this in the following way. First, our approach intercepts some properties of the global XML Http Request object. This ensures that all AJAX network calls pass through Automock. On intercepting an AJAX network call, Automock checks if the response for the particular call is stored in its data file. Automock then checks if there are possible alternate responses. Based on configuration settings, Automock decides which response to return. If no specific configuration is set, Automock returns the default response. If a stored response is found, Automock returns the updated response. To achieve this, it replaces some properties of the original XML Http Request Object. The following properties of the XML Http Request Object are immutable: response Text, ready State, response, status and status Text. Because these properties cannot be modified, Automock has to delete and replace them with the desired values in the XML Http Request Object. This XML Http Request object is then returned to the calling function. Since, the object is identical to the original XML Http Request Object, it works as expected and the application thinks that it made an actual asynchronous network call. In case there is no response present in the data file, Automock passes the call to the original XML Http Request object and makes the actual network call.

These steps ensure that the developer does not need to modify their code at all, while still achieving the functionality required. The mocked response is exactly identical to an actual response, enabling us to make AJAX calls in any preferred way; for example, through the j Query library, directly through an XML Http Request object, or even through any framework dependent-call, such as “fetch” in Backbone.js.

b) Very lightweight

Our approach comprises of just a single JavaScript file which basically comprises of the process outlined above and a socket communication library to interact with the User Interface and the data in real time. It requires no installation and has a very small memory footprint. All the saved AJAX responses are stored in a single flat file which is also minified and serialized. Since an actual server does not need to be run, it also does not consume much CPU memory.

c) Ease of integration

Unlike a traditional server which generally requires an application to be installed and run on one of the ports of the computer, Automock can be included in any web application that uses JavaScript with just a single line of code. As we intercept the native XML Http Request Object, we do not have to deal with issues such as port conflicts. It also does not require any build processes or any other external library to load itself into the system.

d) Flexibility to support as many API calls as required

A developer can mock as many API calls as required. If a mocked API call is not present, Automock forwards the request to the actual backend for resolution. This approach covers a vast variety of use cases wherein the developer can use Automock for only a small module or scale it up and use it for the entire application. This approach also allows the library to be integrated into the project at any stage of the development process. In addition to the above, since we modify the native XML Http Request Object, a user can use any popular library to make network requests such as j Query, Backbone.js, Angular’s $http etc.

e) Automatic capture and mocking of existing API calls

Our approach provides the functionality to capture and mock any existing API calls within the application. It captures all outgoing AJAX requests and maps them to their corresponding incoming AJAX replies. First, it sets up a watch on all AJAX network calls. If any request is noticed by the watcher, it intercepts each returning AJAX network call and stores the response. This stored value is then mapped as the response to the URL for which the AJAX network call was made. Once it has the responses, it extracts each response and transforms the data into a format that the mock server can read. All such transformed responses are combined with our implementation of the mock server and stored in the JSON format. It records the URL, the response, the request type (Such as GET, PUT etc.) and some configuration options. This is serialized and converted into a file that is saved on the developer’s system.

The developer can then simply mock all future calls to the same APIs. Thus, the developer can work without having to constantly query the server, speeding up development since no expensive network calls are necessary.

f) Significant performance boost to unit test case execution

Frontend (and JavaScript) testing is a complex subject with lots of research taking place. Regardless of the desired approach which may be either tool based (Such as Webmate [3] or ATUSA[10]) or automated [2], testing of asynchronous code and especially network requests is challenging.

Developers usually write multiple JavaScript unit test cases to test their modules. Running an entire suite of tests is usually very slow, because a large number of AJAX calls are made repeatedly. In our experience, the bottleneck while running a large number of test cases is the time taken by the network requests. By using our approach, the responses are instantaneous. During our testing, we have experienced a significant performance boost in our unit test cases.

g) No interaction with the database

An important requirement during the develop-pment phase is to avoid adding unnecessary data into the database. To combat this issue, developer teams either use local databases or setup a stage database. Both of these options are time consuming and possibly expensive as well. Since our approach does not make real API calls to the server, it solves this problem without the hassles of setting up a separate database

h) Supports alternate error responses for any API call

A developer must handle error responses during development. It is generally tricky to get error responses out of any good backend in a simple way. Our approach supports returning an error response for an API with some simple configuration settings. A developer can quickly and easily change API responses by either directly modifying the flat file or through the accompanying UI. This approach also helps ensure that a developer has handled all possible cases on the client facing UI.

i) Supports multiple responses for the same API call

Modern web applications now increasingly show different users different data based on the context. For example, when fetching the news feed for a user or fetching list of items for a particular category on an e-commerce site. Automock can be configured to return different responses for the same API call to simulate various situations.

III. Case Study

A version 2.0 prerelease web application was taken up for this case study. The project used an agile methodology and a timeline of about 6-8 weeks. The developers comprised two teams that worked in parallel. One team handled the backend and the other team handled the frontend of the web application. Each sprint was broken down into multiple stories/features being implemented. Here are the various phases we went through during our sprint where we made use of Automock:

a) Step 1: New feature implementation

At this point, both the frontend and the backend developers started development on the new feature. We used Automock quite effectively to make this process much more efficient. The backend developer would create the API stub (The name of the API and what parameters it takes) and use the Automock UI to set the typical response for the API. The frontend developer would then just run the fake server and implement their feature. When the actual API was ready, no more code changes were required for the frontend developer and they could just switch out the mock server for the real server. Since no developer was blocked, both the teams could pick up more features and utilize the entire sprint duration, thus requiring fewer sprints for the same set of features.

b) Step 2: Handling edge cases

Once the frontend developer had finished implementing a feature, they could work on handling edge cases and on handling error cases appropriately. To achieve this, they no longer needed hacks or workarounds. They could just modify the existing mock server response for that API with an error response and continue their development. Since this approach accurately simulates an API call, there is a much better end user experience when things go wrong at runtime.

Table 1: Comparison of time taken while developing for edge cases

	Without AUTOMOCK	With AUTOMOCK
Total Time (sec)	193	8

Notes:

- Time taken without Automock is calculated as: Time taken to modify backend code (~60 sec) + Time taken to build the .war file (76 sec) + Time taken to deploy the .war file (57 sec) = Total Time (193 sec)
- Time taken with Automock is calculated as: Time taken to modify frontend code; that is, changing the configuration variable (~8 sec) = Total Time (8 sec). The time taken to build and deploy the .war file is not required here as no backend changes are needed.
- All times are measured on a typical developer system.

c) Step 3: Adding functionality to pre-existing features

Some pre-existing areas of our code had to be modified to add new functionality. This is where we used one of Automock's best features - Automock can automatically capture and generate mock responses for all existing API calls. We captured all outgoing requests and stored the incoming responses. Since the application now no longer made time-consuming API calls, code edits and unit testing in these areas took much less time.

Results:

Table 2: Comparison of time taken to load four different modules of our application

Time Taken (sec)	Without AUTOMOCK	With AUTOMOCK
Module 1	14.11	0.31
Module 2	18.13	2.90
Module 3	31.63	0.21
Module 4	49.07	0.20

Notes:

- Modules in this table refer to a section/page of our application, each of which loads a different number of asynchronous AJAX calls.
- All times are measured on a typical developer system.

d) Step 4: Third-party services

Our application has dependencies on various third party services. We use these services for authentication, community forums, bug tracking etc. We encountered frequent outages from these third party services, especially on the stage environments. Using

Automock, we were able to mock all the related network calls and responses. Once this was done, we were no longer dependent on the availability of the third party service. This helped us mitigate any delays in development caused by the outages.

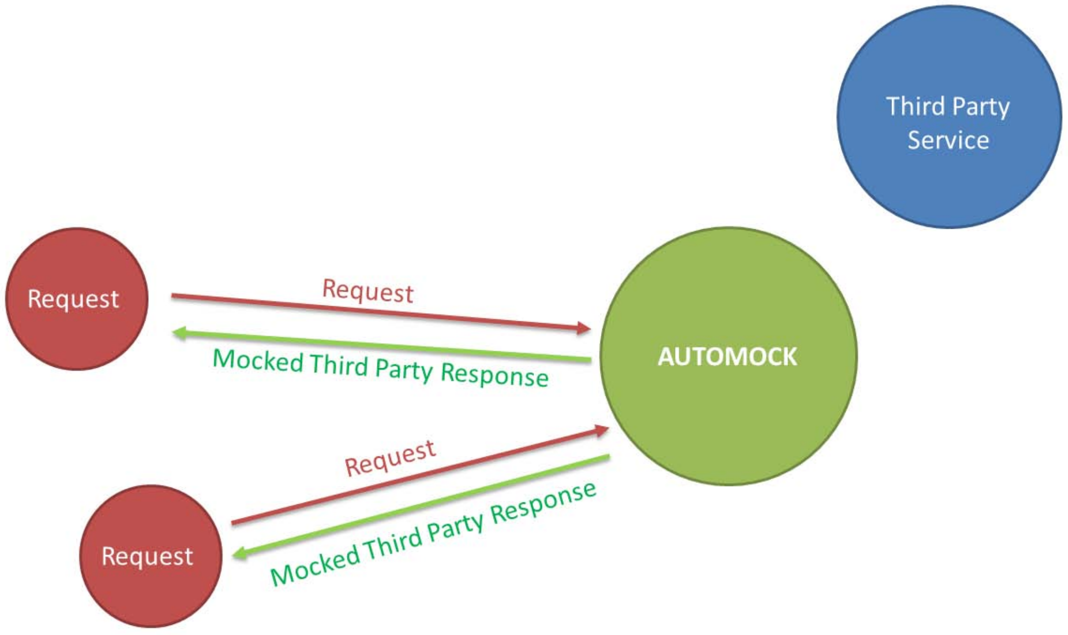

Figure 2: Third-party services

As observed in Fig 2, the third-party service was completely isolated. All requests that were intended for the third party service were easily captured and mocked by Automock. All that a developer had to do was to either let Automock capture a live call or set the response to a particular call manually.

e) Step 5: Unit Testing

Once the frontend developer has finished implementing a feature, they can then write the unit test cases for it. Generally, test cases that make network requests take a long time to complete. Such test cases are also time-consuming to write, since asynchronous logic is hard to implement in most testing frameworks. We have observed that most of the execution time of test cases is taken up by network requests.

Automock helped us solve this problem in a very elegant manner. Since mocked API calls return instantaneously, there was no need to handle asynchronous logic in the test cases. Also, since no expensive network calls were made, the test suite ran significantly faster. This gave us the double benefit of faster test case execution (with no messy workarounds for handling asynchronous calls) and faster test case creation. It also helped us write test cases with real-world data that was static and repeatable. Using Automock, we also avoided polluting the database with junk test data.

f) Step 6: Context based responses

Modern web applications are moving towards context sensitive responses. The same API call can return different responses based on multiple parameters. For example, our website returns different responses based on the credentials of a user. Using Automock, we were easily able to run the application as a different user. We set configuration parameters/flags and ran the application with different contexts. This allowed us to thoroughly handle all the cases that an end user might face, making our application much more robust and user friendly.

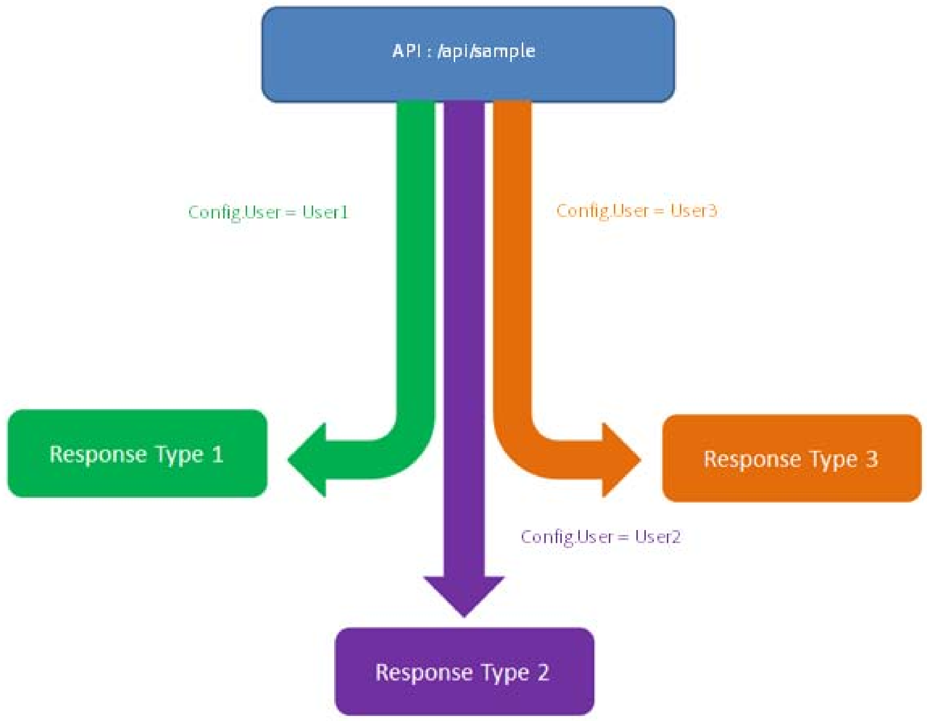

Figure 3: Configuration-based API responses

As seen in Fig 3, a developer would set a configuration parameter to modify the response to an API call. For example, we see that on setting config. user as User1, we get "Response Type 1" as the mocked response. However, on setting config. user as User2, we get "Response Type 2" as the mocked response. We could, similarly, set as many alternate responses as we required using configuration options.

Table 3: Comparison of time taken to load the application as a different user

Time Taken (sec)	Without AUTOMOCK	With AUTOMOCK
User: User1	108.56	3.51
User: User2	112.94	3.51
User: User3	121.27	3.51

Notes:

- Our application has user-specific data. Hence, the time taken without Automock varies for different users.
- The time taken mentioned in this table was the aggregate time taken to load all the four modules mentioned in table 2.
- All times were measured on a typical developer system.

IV. Related Work

JavaScript and Web Development in general are exciting fields for research and development. Our work is focused on easing the experience of web development and testing.

JavaScript application testing is a comparatively recent field due to the increasing size and complexity of modern web applications. More recently, there has been extensive research in the areas of automated testing [12; 13].However, this will still require having to either make the actual network call or write stubbing or mocking logic for the network call. Our approach helps us handle this problem easily and efficiently by mocking the API automatically. Since, the API calls are mocked using our approach, the actual network calls do not have to be made and no extra stubbing logic is required.

Along with research, there are existing libraries and tools to aid web development. Since it is an area of intense activity, there are some libraries already present in this space. In order to adequately put into context the related work in the field, it will be helpful to list down the minimum set of features that we required.

Any framework or library that we use should have a certain baseline of requirements. It should be independent of the development phase (Support use during both testing and development). It should mock network calls without requiring a change in code. It should automatically capture existing network calls as well as allow for the creation of mocks for new network calls. It should support polymorphic responses to network calls. Lastly, it should be lightweight to include and should have zero interaction with the database.

Some of the libraries under consideration by us were:

a) SinonJS [14]
b) Jasmine-AJAX [7]
c) Api-mock [1]
d) Mockjax [8]

a) SinonJS

SinonJS is one of the most popular mocking/stubbing frameworks around. It is great at stubbing and mocking API calls. However, it is limited in its scope as it is a purely testing focused library. Though powerful as a test tool, it requires a great deal of setup and teardown to use in tests. However, SinonJS does not work at all during the development phase.

b) Jasmine-AJAX

Jasmine-AJAX solved one of the most pressing problems with SinonJS – easily mocking API calls. Jasmine-AJAX provides an easily customizable framework to modify the response to a network call. However, it also has a major limitation of only working with the Jasmine testing framework. Similar to SinonJS, this is also a testing focused library and does not work during the development phase.

c) API-Mock

API-Mock is an excellent tool to generate a mock server (running on Express) based on API blueprints. API-mock lets you document your API in the API blueprint format, generates mocks for your routes and sends the responses defined in the API spec. Since API-Mock generates a mock server, it can be used during both development and testing phases. However, it has the caveat of not working well with the existing server. Code changes are required to accommodate the generated API-mock server configuration. Due to this, it was not a good fit for our requirements.

d) Mockjax

Mockjax provides the easiest way of mocking API calls as compared to the other libraries listed above. One drawback of this library is that it is a manual process. The typical workflow for using Mockjax is to integrate the backend code and make the AJAX network call. Then a developer needs to copy the response for each call manually. Then they must transform the response into a Mockjax supported format. Finally, the developer must paste this formatted response into a file and integrate the library.

Though the process seems simple, the time taken to manually add calls using this workflow takes a large amount of time and effort. For a medium to large scaled project, this problem is compounded since a very large number of AJAX calls must be integrated into the application.

A combination of the factors above led to the development of Automock.

There has been some research where the XML Http Request Object is either monitored [16] or encapsulated [11]. To the best of our knowledge, Automock is the only original research paper that overrides a part of the native XML Http Request Object for automating the mocking of network calls. This not only aids in testing but also in development and achieves the goal of removing the dependency between frontend and backend team during agile sprints.

Table 4: Comparison of Automock with other related libraries

	SinonJS	Jasmine-AJAX	Api-Mock	Mockjax	Automock
Support testing and development			✓	✓	✓
Mock without code changes	✓	✓		✓	✓
Support polymorphic responses	✓	✓	✓	✓	✓
Automatic network call capture					✓
Support creation of new network requests			✓		✓

V. Conclusion and Advantages

As we have demonstrated through this paper and through the data provided in the tables, our approach realizes tangible and measurable benefits during development of a web application. It is most effective when interdependent teams are working together. Here are the key benefits:

Makes development sprints more effective by efficiently utilizing developer time

- Speeds up website development by mocking network calls instead of making them every time
- Considerably speeds up test cases
- Aids in quicker development of new features when backend and frontend teams work in parallel

- Helps manage third-party service outages
- Makes development of error responses much more straightforward
- Helps in testing the application with different contexts (Polymorphic API responses)
- Avoids any database interaction during the development and testing phases

References Références Referencias

1. Api-Mock. Api-mock. https://github.com/localmed/api-mock, 2016.
2. Artzi, S., Dolby, J., Jensen, S.H., Moller, A., Tip, F. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, (ICSE '11), ACM New York, 571-580.
3. Dallmeier, V., Burger, M., Orth, T., Zeller, A. WebMate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, (JSTools '12), ACM New York, 11-15.
4. Dinakar, K. Agile development: overcoming a short-term focus in implementing best practices. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, (OOPSLA '09), ACM New York, 579-588.
5. Ganis, M., Maximilien, E.M., Rivera, T. A brief report on working smarter with Agile software develop ment.*IBM Journal of Research and Development, 54* (4), 1 – 10.
6. Garrett, J. 2005. Ajax: A new approach to Web applications. Adaptive path. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/, 2016.
7. Jasmine-AJAX. Jasmine-ajax.https://github.com/jasmine/jasmine-ajax, 2016.
8. Mockjax. Jquery-mockjax. https://github.com/jakerella/jquery-mockjax, 2016.
9. Kosk, A., Mikkonen, T. Rolling out a mission critical system in an agilish way: reflections on building a large-scale dependable information system for public sector. In *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, (RCoSE '15), IEEE Press Piscataway, 41-44.
10. Mesbah, A., Deursen, A.V. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, (ICSE '09), IEEE Computer Society Washington, 210-220.
11. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, (OOPSLA '09), ACM New York, 1-20.
12. Mirshokraie, S., Mesbah, A., Pattabiraman, K. JSEFT: Automated Javascript Unit Test Generation. In *IEEE 8th International Conference on Software Testing, Verification and Validation*, (ICST '15), IEEE Graz, 1-10.
13. Negara, N., Stroulia, E. Automated Acceptance Testing of JavaScript Web Applications. In *19th Working Conference on Reverse Engineering*, (WCRE '12), IEEE Kingston, 318-322.
14. Sinon. SinonJS. http://sinonjs.org/, 2016.
15. XHR. XML Http Request. https://xhr.spec.whatwg.Org /, 2016.
16. Zheng, Y., Bao, T., Zhang, X. Statically locating web application bugs caused by asynchronous calls.In *Proceedings of the 20th international conference on World Wide Web*, (WWW '11), ACM New York, 805-814.