UNIVERSITY OF WATERLOO

Software Engineering

Rich User Interactions without Complex Developer Experiences

Shopify
Ottawa, ON

Prepared by
Justin S. D. Li
Student ID: 20461748
User ID: jsdli
2A Software Engineering

May 12, 2014

Justin S. D. Li
360 Erb St. W.
Waterloo, ON N2L 1W6

May 12, 2014

Dr. A. Morton, Director
Software Engineering
University of Waterloo
Waterloo, ON N2L 3G1

Dear Dr. A. Morton:

I have recently completed my second (2A) work term at Shopify. Please find attached my first work term report: "Rich User Interactions without Complex Developer Experiences".

Shopify is a hosted commerce platform that helps merchants sell online. By heavily emphasizing user experience Shopify has become popular among small to medium sized businesses, and has been able to grow at a very high rate.

As Shopify's customer base and revenue increases, the size of the development team is expanding accordingly. Growing pains, especially increased communication overhead and siloing of critical knowledge, are beginning to become apparent. These effects have been particularly noticed by the Admin Team, as we develop the core of Shopify: the store administration interface for our merchants.

In a collaborative effort of members of the Admin Team, other core teams, and even the CEO, we challenged our current approach to building the admin interface, and designed a simplified system focusing on developer accessibility without compromising user experience. This report describes how we approached the subproblem of website rendering.

I extend my thanks to everyone I worked with at Shopify for their guidance. I learned an incredible amount in my time there.

I hereby confirm that I have received no help, other than what is mentioned above, in writing this report. I also confirm that this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Justin S. D. Li
Student ID: 20461748

Executive Summary

As Shopify grows, communication is beginning to incur more overhead. This has lead to difficulties with scaling the pace of development on the store administration interface (referred to as "the Admin"). Complexities in the architecture made it difficult to understand, and developers looking to implement small features often needed to consult the Admin Team unnecessarily. To mitigate this, we decided to challenge the approach being taken, and come up with a simpler way of building the Admin.

In order to take advantage of the strengths of Shopify's developers, we decided to move page rendering from the client (web browser) to the server. This is more intuitive to most of the development team, who have little experience with bleeding edge client-side JavaScript concepts. To keep the user experience optimal, we came up with a way to update subsections of the page while keeping rendering simple.

For the few remaining interactions that could not be moved to the server, we built a small framework that eliminates a large amount of boilerplate code. Its focus on simplicity makes it extremely understandable, leading it to be easier to use and debug compared to alternatives we explored. Despite its minimalist design, it is able to accomplish the needs of a fluid, dynamic interface, and contains a couple novel ideas that broaden its applications. If its purity continues to be seen as its most important attribute, it will be able to suit the project for a long time.

The solutions we proposed have been working well in practise, and the quality of the result has been exciting. However, as the Admin continues to be developed, the high-level goals of approachability and scalability still need to be kept in mind. They will only become more relevant as the size of the Admin increases, and their attention will be an important factor in the long-term maintainability of the product.

Table of Contents

List of Figures

1 Introduction

Shopify is a commerce platform that helps merchants sell online. As a fully hosted solution, it provides a simple, intuitive experience for first-time and experienced merchants alike. Powering over 100,000 stores [1], Shopify's merchants have sold over $4,000,000,000, attracting names like BlackMilk, The Chive, and Wikimedia. Having raised a $100 million Series C funding round in December 2013 [2], Shopify is one of the fastest growing companies in Canada. Each year, hiring continues to double, bringing the company over the 400 person mark in early 2014 [1].

This aggressive growth is not without its challenges. As the development team expands, difficulties with coordinating large-scale communication are leading to cases of information siloing – the isolation of knowledge to a subset of the team. The Admin Team, who build the store administration interface for merchants (dubbed "the Admin"), are among the most heavily impacted by this phenomenon.

Since Shopify's customers use the Admin to control every aspect of their store, the majority of other projects in the company require changing it to accommodate new features. It would be unscalable for the Admin Team to be responsible for every feature, so a large part of the overall development team needs to know how to make the changes they need. However, due in part to the architecture of the Admin, it was very difficult for most developers to quickly become productive.

To mitigate this issue and build a better base for future development, we decided to challenge the fundamentals from which it was built and rethink our approach entirely. This report covers the strategy we chose for implementing the user interface, specifically the two highly related subproblems of page rendering and dynamic interaction. It is intended for the Shopify development team, and requires a limited understanding of the basic concepts of web development to be useful.

2 The Architectural Flaw

2.1 Rich Web Interfaces and the Single Page App

Since the advent of HTML in 1993 [3], there has been, for the most part, one way of building websites. A web browser asks the server to send back some HTML, parses and renders it, then stops working. When the user clicks on a link indicating they would like to navigate, the browser repeats the same process to fetch and display the next page (shown in Figure 2-1).

In many cases this process is appropriate, but as websites began to imitate and replace desktop applications a new approach was necessary to match the fluidity and responsiveness of native experiences. Instead of relying on the browser's default behaviour, people began using JavaScript to dynamically update smaller sections of the page without requiring a full reload [4].

A "single-page application" (SPA) takes this idea to its full extent, by generating all HTML in the browser with client-side JavaScript and only ever loading *data* from the server (see Figure 2-2). Logic for displaying data, and to some extent manipulating it, sits entirely in the client.

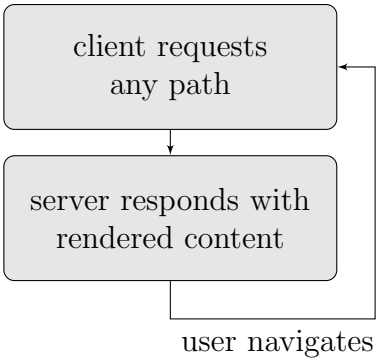

Figure 2-1: Traditional navigation flow

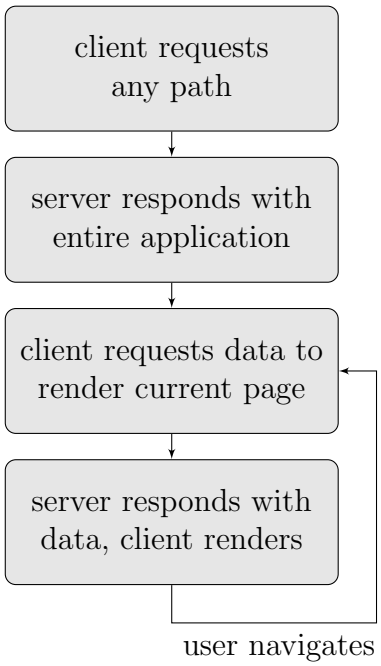

Figure 2-2: SPA navigation flow

The single-page app method has a number of advantages due to the native approach it takes to rendering. Navigation between pages can be faster, since the HTML templates needed to render them are already loaded into memory. Since the client generates the HTML, it can dynamically alter the page with new data without requiring both the server and client know how to render the same data. Perhaps most importantly, since the client transfers raw data, the communication method it uses can also support other clients (e.g. mobile apps).

For primarily these reasons, in 2010 Shopify decided to create a framework (called batman.js [5]) to rebuild the Admin as a single-page app. Doing so greatly improved our data communication layer, and helped bring the user experience to new levels.

2.2 Motivation For Leaving the SPA Architecture

Despite the strengths of the SPA approach, by the time the new Admin was fully implemented it had become quite complex. Since so much responsibility was moved to the client, a whole layer of duplication was created, and sometimes subtle differences introduced obstinate issues. Both the client and server needed to know that an Order contains a tree of Transactions, which can be Refunds or Captures, that they interact in certain ways, and how to create them. This is clearly not ideal.

Developer approachability suffered (simple tasks became difficult for unacquainted developers), in part due the unconventional approach and in part due to the breadth of the Admin. Shopify's platform is extremely extensive, and what worked well for smaller applications was no longer scaling for a much larger one.

Since the Admin is how merchants control all aspects of their store, most new features or products need to modify or integrate with it. As a result, nearly every developer has to write code for it at some point. The Admin Team thus became a bottleneck, since the majority of developers outside the team needed to consult us to do pretty much anything. This slowed development considerably, since it compromised the productivity of both those needing help and the people helping them.

In addition to this, the architecture of batman.js and how we were using it had significant performance issues. For shops with a great many products or orders, some (especially mobile) devices had a hard time rendering certain pages. This is not only aggravating to customers with low-end devices, but also subliminally affects everyone's perception of the product's quality.

3 The Simpler Approach

Our mantra from day one was simple:

Easy to get right, hard to get wrong.

This was motivated by the tendency of large projects to deviate from their initial purity when their underlying design doesn't enforce simple, correct patterns. This philosophy doesn't just imply the design should be resilient, but also that it should be easy to understand and simple to debug. The high-level goals of this project were derived from this idea:

- Approachability. Any developer must be able to make the changes they need without getting lost in the code. When the existing code is confusing or complex, people tend to produce more complex code.
- Scalability of knowledge. As the company grows, the number of people pushing code is growing at a dramatic rate. The underlying platform needs to support this without becoming bottlenecked by isolated knowledge and slow communication.
- Scalability of performance. As the number and size of merchants using Shopify grows, the platform needs to keep getting faster and better. To accomplish this, it needs to be easy to make Shopify faster.

With these ideas in mind, the project was required to produce a new Admin that was functionally and visually equivalent to the old. Since the previous approach was based around the single-page app idea, it used incompatible patterns for two important tasks that we needed to address in the new version: how pages should be rendered, and how any subsequent dynamic interactions with the user should occur.

3.1 Page Rendering

As mentioned in §2.1, HTML has been rendered by browsers for a long time. As a result, they have predictably become very good at it. We decided to leverage this by moving back to having the server generate chunks of HTML rather than having the client fill in the data itself.

Shopify has always used the Ruby on Rails framework (usually called Rails) on the server, as it accelerates development greatly by providing a set of strong conventions [6]. Most developers at Shopify are well-versed in Rails, so moving closer to the conventions it sets improves the team's ability to speak in common terms and iterate quickly.

Server-side rendering is easier and more intuitive for Rails developers, since it is rooted in the way websites have always been built. Indeed, when we described our plans to any developer who was not at some point part of the Admin Team, we were met with unanimous agreement that development would be faster and easier.

Besides approachability, this also gives us more control over scaling the product as load grows. Instead of requiring rendering to be instant on every possible device, the server code runs on machines controlled by us so we can optimize them for their workload. To put it simply, this lets us make the code faster or *make the servers faster*. For a company growing at such a rapid rate, simply adding more servers can sometimes be the cheapest and most reliable way to scale (see Appendix A).

The final major benefit is that the layer of duplicated logic in the SPA approach is not present. This is crucial to reducing complexity that would hamper long-term maintainability.

3.1.1 Partial Updates

Despite the bold declaration that the old ways of rendering are not, in fact, inferior, we still require an incredibly seamless user experience. We cannot get away with only ever redrawing the whole page, as there are some cases where subsections need to be updated independently. This occurs whenever the user may have edited some portion of the page, but then performs an action that needs to re-fetch data from the server.

For example, Shopify allows merchants to set up a blog on their store, and their customers may comment on articles they post. On the article editing page, merchants may moderate and delete comments without needing to navigate away. Deleting a comment always requires re-fetching and displaying the comment list, so that the correct limit of 50 comments is always displayed. However, before performing the deletion, they may have changed other parts of the post (e.g. the title or description), so actions upon comments may not refresh the whole page or those changes would be lost.

In accordance with the overall project requirements, we wanted to come up with a

solution for this that would be nearly impossible to implement wrong. After deliberating on the matter for some time, we realized a crucial insight. Despite only needing to replace a subsection of the page, it would be simplest to render the entire page on the server and pick out the sections needed from the response. This ensures there is only one render path, but still allows for correct and snappy-feeling updates.

Besides being utterly simple, this approach has the added benefit of being able to update multiple portions of the page at once. Since network latency – the time between the client sending the request and the server receiving it – is a major factor in the total delay a user experiences, cutting down to a single request helps minimize it.

However, it relies upon the important assumption that the overhead of rendering the full page rather than the partial page is minimal. For the Shopify Admin, this turned out to be true almost all of the time, since our partial updates usually needed to replace the majority of the page (like in the comment list example).

3.1.2 Partial Rendering Endpoints

An alternative to rendering the whole page every time is to create a new route that just renders the comment section. For example, if the current path is `/articles/143`, then `/articles/143/comments` would return just the HTML needed for the comment list. An action updating the comment list would fetch that URL and replace the current comment list with it. This works, but has several notable downsides:

- The server now needs to know how to render one thing in two ways. This translates to increased maintenance overhead in ensuring both are correct.
- The client needs to know where to look for every resource that needs updating. This is duplicated logic again, which is another maintenance hassle.
- For every additional way of rendering a resource, the server needs to duplicate the logic for loading the backing data.
- Sometimes multiple sections need to be updated at once. This means there are some cases where there need to be a large number of new routes created for one page.

In a couple rare cases in the Admin this approach does fit better, but overall we found any performance gain it might have did not justify the added complexity.

3.2 Dynamic Interactions

In addition to partial page replacement, there is one other crucial component to building rich user interfaces. People expect applications to react to their input, providing them with an indication they are doing the right thing. This is especially important for implementing forms that need their input validated in some way. For example, Figure 3-1 contains some inputs that have maximum lengths.

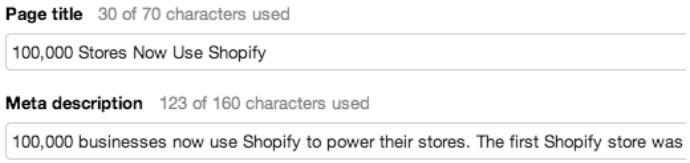

Figure 3-1: Example case of dynamic form elements in the Shopify Admin

Traditionally, websites implement this by allowing the user to enter arbitrary text, and then returning an error message after they save if the input was invalid. This is tiresome for the user, since it may take multiple tries to come up with valid input, and the process of saving and waiting for the server is slow and cumbersome.

To reduce hassle for the user, the wait time can be mitigated by performing the validation directly in the browser. Figure 3-1 demonstrates how displaying the length of the currently entered text simplifies the experience. Since the user immediately sees how their input is faring, they can immediately begin to formulate an allowed version without moving onto the rest of the form and losing context.

3.2.1 Data Binding

We decided our ideal solution would be to simply declare what the *result* should be, and let something else take care of making the correct behaviour occur. This general idea is not novel when applied to visual interfaces, and is usually referred to as "data binding" [7].

Figure 3-2 shows how we might write binding-based HTML for the form in Figure 3-1. This code declares that the text input should be bound to the `title` variable, and that the text in the label should be bound to the length of `title`. The end behaviour is that when the user types within the input, `title` should be updated with the new value, and the length should be displayed accordingly.

```
Page title: <span bind="title.length"></span> of 70 characters used.
<input type="text" bind="title">
```

Figure 3-2: HTML implementation with bindings

This is concise, expressive, and most importantly requires writing zero lines of JavaScript. Even without former knowledge of HTML, this binding representation should be clear to most people. Unfortunately, there is no built-in construct for this in HTML, so we needed to create our own implementation (alternatives discussed in §3.2.3 and §3.2.4).

3.2.2 Implementation

The binding system we created has two roles: to listen for and process user input, and to propagate data back out to the interface. The intermediary between these two processes is a single object where dynamic data is stored ("storage"). The flow of data is visualized in Figure 3-3.

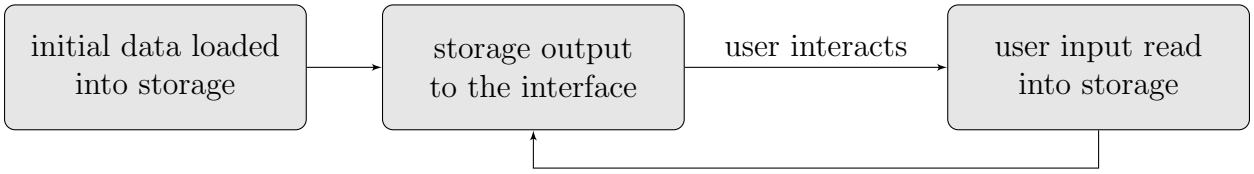

Figure 3-3: Flow of data in the binding system

To capture the user's interaction events (key presses and mouse clicks), event listeners need to be attached to each relevant interface element. Web browsers represent these elements as trees of HTML nodes. In Figure 3-4, an example tree is shown that demonstrates the structure of Figure 3-2.

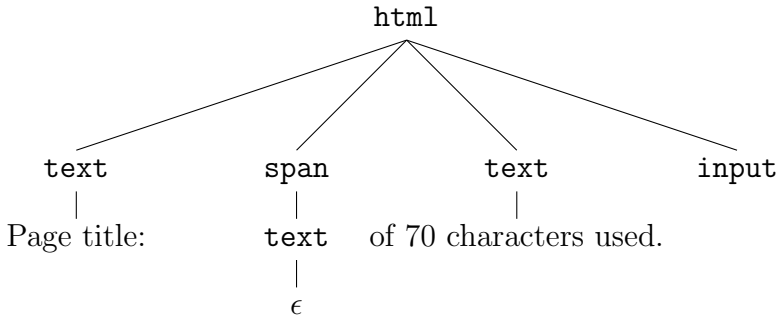

Figure 3-4: The tree representation of Figure 3-2

To attach the necessary input event listeners, all nodes with a `bind` attribute need to be located and processed. To do this, we used a depth-first search that traverses the entire tree while looking for relevant nodes. On each node, an event listener is attached, which when triggered reads the node's new value into the binding storage, and queues the binding system to propagate data back to the interface.

To support using bindings without previously declaring variables as in Figure 3-2, we needed to instantiate them on the fly by loading data from the HTML. We used the same event-attaching traversal to do this step before reading and displaying data for bound nodes. This is a novel approach to implementing a binding system, and allows it to be used in a wider range of situations than requiring all data to come from JavaScript.

To be useful beyond trivial examples, the binding system needed to support transforming data before displaying it. To allow this, we decided to take advantage of JavaScript's dynamic nature by running `bind` expressions as JavaScript themselves. This takes a whole two lines of code to implement and runs extremely quickly, while a custom parser could be thousands of lines of code and not provide any performance improvement. A JavaScript-based binding is illustrated in Figure 3-5, which will output `title` with no leading or trailing whitespace, and with all birds replaced with cats.

```
1 <label bind="title.trim().replace(/bird/g, 'cat')"></label>
```

Figure 3-5: A binding with dynamic JavaScript

To reduce expensive interface-updating operations, we kept the previous output of each binding around in memory. On each iteration of the output propagation step, the new value is computed, but no updates will be made unless the output actually changed. Implementing this simple augmentation dramatically improved performance, since browser limitations cause updating the interface to be slow even if nothing really changed.

At each step, choosing the simplest option allowed us to create a system so small that it can easily be understood and debugged by developers. At just over 200 lines of code (with comments), it fits the project's goals by being approachable and practical. Performance matches any other binding system we tried, and its small size means it will be maintainable in the long term.

3.2.3 Dynamic Behaviour without Bindings

Rather than building a binding system, the most obvious way to accomplish dynamic behaviour is to write procedural code that alters the page exactly as required. Figure 3-6 demonstrates how we might implement Figure 3-1 without bindings:

```
Page title: <span id="title-count">0</span> of 70 characters used
<input type="text" id="title">
```

Figure 3-6: The HTML of Figure 3-2 without bindings

Then, we would make it react dynamically by running the JavaScript in Figure 3-7. If you don't know JavaScript, you can read the comments in grey.

```
// Get the elements we need to read and update
var titleNode = document.getElementById("title");
var countNode = document.getElementById("title-count");

// Whenever the content of the title input changes
titleNode.addEventListener("input", function() {
  // Update the content of label to display the length of the input
  countNode.textContent = this.value.length;
});
```

Figure 3-7: The JavaScript needed to drive the interaction

There is a comparatively large amount of code here to accomplish an extremely common and straightforward task. If even the simplest of interactions requires this much effort, a very large application like Shopify could quickly become unmaintainable. In addition, the HTML and JavaScript are very tightly coupled, meaning changes to one side would require corresponding changes to the other.

3.2.4 Alternative Binding Systems

Before deciding to build our own binding system, we considered a number of excellent external open-source implementations:

- AngularJS [8] is a powerhouse framework made by Google, that includes everything you need to make a single-page app (and more). Their binding system is clean and fast, but the size and relative inseparability of the library components makes it unsuitable for standalone use.

- Knockout [9] is a lighter-weight library that only implements bindings. However, it requires declaring every possibly bound variable before it's used. This means you still need to write JavaScript to implement common interactions, and the elegant HTML in Figure 3-2 is not possible.

- The binding system in batman.js works well, and the Admin Team would already be very familiar with it. However, it is so entangled in the rest of the framework that an extraction would be infeasible. In addition, its lacklustre performance was one of the reasons we turned away from it in the first place.

After evaluating these options and others, we realized none shared our goal of ultimate simplicity. They either required jumping through hoops and writing code that relies heavily on the specific implementation of the framework, or were monoliths built with thousands of lines of code. Thus, we challenged ourselves (and succeeded) to implement the simplest binding system possible, while still covering all of our possible use cases. At the time of this writing, it is still just over 200 lines of code, with comments. Any developer is able to dive in easily and understand how the whole thing works, which is a rare property for a library.

4 Conclusions

The biggest goal of this project was to create a simple experience for developers. By carefully examining the characteristics of a great user experience, we were able to come up with patterns that not only reduce complexity, but also are able to handle the interface designs we think are optimal.

By moving the rendering back to the server, we were able to cut out a large layer of extreme complexity. This takes advantage of the number of strong Rails developers at Shopify, by sitting closer to the intuitive methods commonly used for web development. To allow server-side rendering to work with partial page updates, we took a novel approach of rendering the entire page and selecting the desired portions to be replaced. This is impossible to get wrong, fast, and works for every case the Shopify Admin needs.

For the remaining dynamic user interactions, we built a binding system to reduce the amount of interface code. By enforcing understandability above all else, the system we built is easy to use, improve, and debug. Alternative binding systems were judged to be unfitting for this project, since they contained unnecessary features or were inextricably linked to monolithic frameworks. Since the new system is so small, it will be easily maintainable even if the initial designers move onto other projects.

In practise, the solutions we picked turned out to be successful in meeting the project's goals. Developers outside the Admin Team were able to build new sections quickly, producing good implementations without much communication overhead. The project is still underway, but the results are promising so far.

5 Recommendations

The goals of this project are not temporally local to it, and should be kept in mind for the lifetime of the Admin. Simplicity will only become more important as the size of the Admin continues to grow. If any earlier designs turn out to be causing unnecessary complexity, they should be carefully reconsidered before they become hard to change. No decision is sacred.

The binding system we built should remain aggressively simplified, and new features should be kept to a minimum. Its key strength lies in how concise and understandable it is, rendering it trivial to debug. The worst case is when developers are unwilling to approach a problem because the underlying system is too complex to understand.

Even with the simpler approach to development, efficient large-scale communication of knowledge still needs to be solved. This is a challenge every growing company faces, and a good solution would have a tremendous impact on success. I have no answers here, but exploring this area is likely to pay off.

References

[1] *The About Us Page of Shopify*, http://www.shopify.com/about (current May 2014).

[2] T. Lükte, *Shopify raises $100 million*, http://www.shopify.com/blog/10780917-shopify-raises-100-million (current May 2014), 2013.

[3] T. Berners-Lee and D. Connolly, *Hypertext Markup Language (HTML)*, http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt (current May 2014), 1993.

[4] M. Galli, R. Soares, and I. Oeschger, *Inner-browsing extending the browser navigation paradigm*, https://developer.mozilla.org/en-US/docs/Inner-browsing_extending_the_browser_navigation_paradigm (current May 2014), 2003.

[5] *batman.js*, http://batmanjs.org/ (current May 2014).

[6] *Ruby on Rails*, http://rubyonrails.org/ (current May 2014).

[7] Microsoft, *Data binding overview*, http://msdn.microsoft.com/en-us/library/ms752347.aspx (current May 2014).

[8] *AngularJS*, https://angularjs.org/ (current May 2014).

[9] *Knockout*, https://knockoutjs.org/ (current May 2014).

Acknowledgements

This project started as a collaboration between Kristian Plettenberg-Dussault (Admin Team Lead), Tom Burns, Arthur Neves, Tobi Lükte (CEO), and myself. Some of the ideas in this report are based on their contributions.

I would like to thank Petra Watzlawik-Li for her proofreading help.

I used the uw-wkrpt document class for LaTeX written by Simon Law to typeset this report. I extend my thanks to him and the project's other contributors for making it so simple to use.

Appendix A Comparison of Approaches to Scaling

It might seem counterintuitive that throwing servers at a problem is sometimes the best way of solving it. However, consider the following scenario:

Cost of one server: $1,000
Number of servers: 100
Average developer pay: ~$50/hour

Since there are 100 current servers, adding one more would increase maximum throughput by 1%. By 200 servers, adding another would still be an increase of 0.5%. A skilled developer would have a very hard time consistently increasing throughput by 0.5-1% in only 2-3 days each time. Of course, a low quality codebase may have many easy ways to improve performance by huge factors, but this is not usually the case for well-designed applications.