Real-Time Frameworks That Are App-ifying The Web

Studies show that even a fraction of a second delay reduces user engagement and site revenue. As a result, the "refresh" arrow is going the way of the dodo, and real-time data update is being transformed from a convenience feature to a necessity for the most competitive sites.

By Peter Baumgartner

Current server-side web frameworks, such as Django and Ruby on Rails, grew out of a need to make common development tasks (e.g., user authentication or storing and querying data from a database) more consistent, less error-prone. Now the same need is leading to new generation of real-time web frameworks running on the client side in a desktop browser. It's part of a larger movement to create sites that work more like native applications.

Avoid this app-ification of the web at your peril!

How Real-Time Sites Are Different

Traditionally, webpages are displayed through a request-and-response cycle: Enter a URL and the browser sends a request to the server, which responds by sending back the page for display. The process works well for informational websites that don't require a lot of user interaction. But with the spread of smartphones and tablets, we've come to expect more responsive apps, and the web is no exception.

But modifying the classic request-and-response cycle to work with parts of the pages, it's possible to create page sections that respond immediately to user input. Single-page applications, as they're called, can be updated and modified in real time. One obvious technical challenge: Interactivity goes both ways, which means that applications need to be ready to accept new data from a server without ever making a request.

Why Real-Time Is Useful

Take a web-based chat room. Utilizing real-time frameworks, we can send new messages from a user to the server, and then push them out to all the other users with very little work. With previous solutions, such as polling or long-polling, the browser was unable to wait for new messages from the server to be pushed to it. Instead, the browser had to ask constantly if there were new messages. The implementation could be tricky and there was no way to prevent unnecessary requests to the server, resulting in greater overhead and delays in data reaching all clients.

WebSockets enable servers to create a continuously open socket connection for bi-directional messaging and updating. Once established, the server is able to initiate communication to the browser. Messages are sent as soon as they are received, instead of waiting for the browser to poll. Polling systems were always a bit of a hack around the existing technology limitations, but WebSockets give us a clean way to handle the problem. Also, by removing the delays of polling, we can handle a whole new class of problems in which real-time information is critical (stocks, auctions, etc.) and a few seconds' delay can be costly.

Designers Will Need To Adapt

Web applications aren't new. For example, Microsoft has offered a web-based version of Outlook for years, and Google Docs has passed its five-year anniversary. These sites were massive engineering feats because the whole stack had to be built from scratch. Only big companies and heavily funded startups could compete in this realm. Now, the combination of improved web browser technology and open source web frameworks makes it possible to build simple chat applications in fewer than 100 lines of code. This is a good thing, but developers will need to adapt.

Previously, you could get away with a server-side developer and a designer who knew some HTML and CSS and could inject some interactive flair with a few jQuery effects; no longer. The development techniques we've applied to server-side code (modularized, decoupled MVC-style code) will be expected on the client side.

For designers squeaking by with barely enough knowledge to build a static webpage, it will be tough to keep up. The new paradigm will be more familiar to server-side developers, but in some ways it will seem quite foreign. For instance, unlike server-side code running in a controlled environment, client-side code needs to run in many different browsers, each with its own JavaScript run-time environment. Also, client-side development is typically event-driven, such as mouse clicks or data submission, Sites, like Hulu, use Backbone.js to allow users to quickly navigate through "pages" without actually reloading all the content from the server on every click.

The Server-Side Problem (And Some New Solutions)

The client is only half the equation. Developers still need to get the data on the server down to the client, as well as handle secure transactions, such as user authentication. Adopting a new server framework or adapting your existing infrastructure to handle real-time connections is not a trivial undertaking. Companies must decide how to push data in real time from their own server stack. And to accommodate such backend changes, they must also redesign aspects of the frontend. Commercial services, such as Pusher, let you piggyback off their real-time infrastructure, making it easier to add real-time features such as multiplayer games, backend dashboards, and multi-user collaboration.

Firebase provides features to companies that don't want to manage their own server stack.

But as some see it, none of this addresses the real problem: We're trying to force a square peg into a round hole. These contrarians argue that our existing web frameworks weren't built to run real-time applications, which is a valid point. Which is why yet another new breed of frameworks optimized for single-page web applications is being built. Some of the frontrunners include Meteor, Derby, and SocketStream, and a lot of eyes are on them to see if they can deliver on their promise. Meteor alone secured $11 million in VC funding to accomplish its goal.

Time will tell if these trailblazers will create successful frameworks. This is still experimental, and so there are bound to be some false starts, but if they can quickly correct course without accruing legacy code debt, I think they have a chance. If they stray too far in the wrong direction, it's more likely that new projects will come along and, learning from their mistakes, leapfrog them in popularity.

Learning To Live With JavaScript

Real-time websites aren't without their problems. JavaScript is the only scripting language available to all web browsers, and the rise of single-page applications has significantly increased its usage. With the advent of Node.js, JavaScript is also being used to write server-side code.

Unfortunately, JavaScript is a quirky language which developers either love or hate. The haters have been able to skate by with minimal knowledge of the language, but the freedom to hate JavaScript is fading fast. It will be hard to survive, much less to thrive, in a world of real-time web applications without a thorough understanding of JavaScript.

Real time is already here. Those hundreds of small icons on your smartphone are a never-ending showcase of our interactions with real-time messaging, monitoring, and gaming. It's possible that soon, we'll be reminiscing about the old days when we loaded full HTML webpages from a server—just like we reminisce about having a telephone operator connect our calls. Coders, clients, tech leaders, and web users in general: Are you ready for this next version of the web?

Peter Baumgartner is the founder of full-service web studio Lincoln Loop, a premier Django development and consulting agency which has built sites for National Geographic, PBS, Evite, and others, as well as online software products like Ginger. Peter is most proud of Lincoln Loop for giving away over $12K to charity last year. He now lives in Mexico with his wife and two children and enjoys surfing in his free time. He welcomes anyone to reach out to him on Twitter or Google+.

[*Image by Irargerich on Flickr*]

April 14, 2013 | 8:55 PM

ADD NEW COMMENT

SIGN IN

Type your comment here.

0 COMMENTS

No comments yet. Be the first!