Evaluating web frameworks

Abundance

Stop re-inventing the wheel and build your web applications with the excellent tools already available. *By Kurt Seifried*

One theme I've noticed in many large web applications is badly reinvented wheels. I suspect a lot of this is caused by the "not invented here" syndrome or by developers who want to avoid external dependencies (portability is nice). The problem is that virtually all web applications have a rather complex set of requirements and security needs that often are not implemented well (if at all). And, a lot of us who have been programming web applications for more than a decade might still be a bit mentally stuck in the 2000s, when a little HTML and some form fields were all you needed to make an "interactive" site that actually worked quite well. Conversely, I can't help but think newer programmers aren't aware of all the problems already discovered and solved in frameworks during the past decade and a half.

Why Frameworks?

One compelling reason to use frameworks is that they not only do a lot of the heavy lifting, the better frameworks do it properly. I say "better frameworks," because at this point there are literally thousands, and many are terrible. Additionally, the more popular frameworks, like Django and Ruby on Rails, have tens of thousands of add-ons (Ruby Gems, Django plugins, etc.) that can be used to provide functionality. Plus, someone else maintains the code for you: win-win! The downside of these frameworks is that you have to build your applications using their idioms and design patterns: If you haven't read *Design Patterns* [1], about designing object-oriented software, you might want to buy a copy.

MVC and REST

One popular design pattern with web frameworks is the MVC (Model, View, Controller) pattern (Figure 1). In a nutshell, you have a *controller*, which sends commands on the basis of either user actions or other external or internal events, like an order generation; a *model*, which represents the system, essentially a large state machine; and a *view*, which is output sent to the user or other systems. One reason this design pattern is popular is that it supports stateless protocols (like HTTP) rather well, whereas many traditional design patterns assume a fully stateful system.

Another popular pattern is the REST (REpresentational State Transfer) pattern, which maps directly to HTTP 1.1 and loosely to many back ends, like SQL, NoSQL, cache engines (e.g., memcached), and so on. REST makes several assumptions, including a *client-server design*, which is pretty much the rule for all web-based components, including back-end elements like SQL/NoSQL. It also assumes a *layered system*, which is generally the rule for web-based applications; *caching* – again, something generally done at most levels in web apps; and *statelessness*, also typically the rule for web apps and back-end components.

Picking a Framework

An obvious constraint on picking a framework is picking one that supports your language(s) of choice. In the Ruby world, you're pretty much always going to pick Ruby on Rails [2], and for JavaScript, it'll be node.js. In the Python world, two more popular options are Django [3] and Pylons [4]. Perl and PHP have a lot of frameworks and application-level packages; I think the amount of choice has prevented one or a few from becoming the "standard." For Java, your best bet is going to be JBoss and the related technologies. For traditionally compiled languages like C and C++, you also have many choices; however, picking C and C++ for web-based projects will generally be painful.

Extending a Framework

The chances your framework will have all the capabilities you need are slim; your application likely will need something not provided directly by the framework (e.g., unit testing; messaging; interfaces to other APIs and systems, like cloud providers and Google; etc.). These framework gaps are one reason Ruby on Rails is so popular: It has more than 45,000 Gems covering pretty much everything you can imagine. Python has PyPI (Python Package Index), which is a collection of more than 24,000 packages – one of my favorites is BeautifulSoup, a great HTML parser.

When picking a language and framework, don't forget to take the ecosystem into account. Although I hate to use that word, it's appropriate here: Being able to pull in high-quality third-party libraries and plugins will reduce development

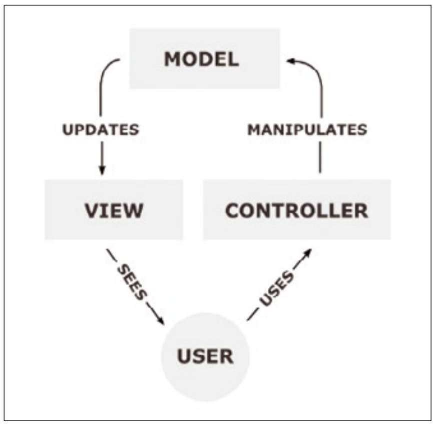

Figure 1: The Model-View-Controller process.

KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

time significantly. Conversely, using unmaintained, badly documented, or otherwise poor-quality libraries and plugins will cause a great deal of pain and increase development (debugging and bug-fixing) time significantly.

Accounts and Authentication

One area in which many applications are weak is in the handling of accounts and authentication. Considering how central this is to most applications (especially if you want to sell something or restrict access to resources), it can be a significant problem. When picking a framework, make sure it supports proper account management: Can you disable an account, as opposed to removing it entirely? Can you create groups and delegate administration? Can you give customers control over their own users?

One popular web application treats accounts as if they are owned by the user, so that even if you, as the customer, add 20 users to access your site, you can't modify their accounts. You can't, for example, disable their account, change the password, add their name or phone number, and so forth. This means you have to chase users around and get them to set up their accounts properly, and for password resets, the user has to contact the provider to do it. You can do nothing to modify the account. Other applications I have encountered do not allow you to list the resources a user account has access to; instead, you have to go through all the resources and check the permissions applied to them, hunting for the permission(s) you want to remove.

Not everyone makes the same assumptions when it comes to account management, so make sure it will work for you. As far as authentication goes, most support username and password, most support external providers like OpenID or OpenAuth, and most can support some form of two-factor authentication. However, very few support client-side SSL certificates or Kerberos, for example. Again, make sure you know what you'll need so you don't paint yourself into a corner.

Logging and Debugging

Another significant weakness in many web frameworks is logging and debugging. Most frameworks provide basic logging (requests, who made them, return status, etc.), but few will allow you to trace a single request easily through multiple subsystems.

For example, a request might first be handled for URL redirection before being passed off to a URL handler. This handler might in turn handle the request through a cached copy, or it might create a new reply, touching multiple other systems (authentication, data storage back ends, caching layers, etc.). These back ends in turn could trigger additional actions. If you were to pass a request ID with subsequent queries (and log everything), you could in theory ultimately see what everything is doing.

Debugging is another sticky problem, especially on production systems. Simply enabling debugging for all requests on a major site might collapse the system (because of logging requirements), so you need to determine whether the framework allows you to enable debugging selectively on a single server and steer all testing requests to it or whether can you enable detailed logging for a specific user or IP address.

Conclusions

Besides the issues I've already covered, you also need to consider the following points:

- Data storage and caching – Things like Python `pickle()` are not safe; use JSON.
- Session management – Are sessions secured against hijacking?
- CSRF – Protection from cross-site request forgery.
- Input and output validation – XSS, SQL injection, etc.
- Clickjacking protection (e.g., `X-Frame-Options` [5]).
- Site changes/URL redirection.
- Websockets – Are they safely supported?

Ironically, many of these popular web-based frameworks have grown out of large web projects at companies that built in-house frameworks and then released them publicly.

As a rule of thumb, however, unless you have a minimum of several dozen dedicated web software programmers that are really good, you should not be building your own framework. Instead, you should find a framework that works for you and start using it. Of course, there will always be exceptions even to this rule. ■■■

INFO

[1] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[2] Ruby On Rails Security Guide: *http://guides.rubyonrails.org/security.html*

[3] Security in Django: *https://docs.djangoproject.com/en/dev/topics/security/*

[4] Pylons: *http://www.pythonsecurity.org/wiki/pylons/*

[5] The X-Frame-Options response header: *https://developer.mozilla.org/en-US/docs/The_X-FRAME-OPTIONS_response_header*