Compared 15 web application frameworks across 10 features

* AngularJS
* Apache Tapestry
* Apache Wicket
* Catalyst
* CodeIgniter
* Django
* Flask
* Grails
* Kepler
* Laravel
* Meteor
* Pyramid
* Ruby on Rails (RoR)
* Spring
* Symfony

These frameworks expand many different applications; some are used primarily for mobile apps, some may be used for desktop applications, and others are intended only for web application usage. They also represent many different languages, some we already have experience in and others that may require a steep learning curve should we choose to utilize them; between all fifteen, there is Javascript, Java, Perl, PHP, Python, Groovy, Lua, C, and Ruby. All are open source and free to use under license.



Green = +1, yellow had no effect, and red = -1 from green score

Narrowed down to the frameworks with scores of 8, 9, or 10 – leaving 9 frameworks for further examination.

We eliminated two more (Flask and Angular.js) by narrowing our criteria down to full stack frameworks—Flash is a microframework, which works well for smaller size projects, and Angular.js is primarily a front-end framework, not expansive enough for our purposes—leaving 7 frameworks.

We’ll look at those more in depth here, ultimately ranking and eliminating them to determine which framework we use.

* CodeIgniter
* Django
* Grails
* Laravel
* Meteor
* Ruby on Rails (RoR)
* Spring

**CodeIgniter**

**Pros:** PHP. Very small download size (2MB). Encourages but doesn’t enforce MVC structure. Built-in protection against CSRF and XSS attacks. Community-developed project, with GitHub base. Fast to implement and easy to learn. Can create your own libraries, easy to link to 3rd party functionality. Not a lot of overhead, very lightweight. “Perfect for simple services where a large workload is not expected.” Gentle learning curve. Rapid development. Easy to set up. Clear documentation. Supports both web page and database caching. Built-in security features. Easy to extend and customize.

**Cons:** Doesn’t support namespaces. Doesn’t support middleware. The next version (v4) will not be backward compatible with the current version. May not be able to support a high workload. “Does not have enough functionality for quickly developing larger web applications.” Does not come with any components to build GUI elements (not necessary for web-based application). Does not emphasize maintainability. Lacks exhaustive libraries.

**Django**

**Pros:** Good ecosystem. Python, we have experience in that. Speeds up the execution. Easy to reuse code. Generates an administrative interface facilitating website updates, moderation of comments, news updates, theme activation, etc. Easy to extend. Huge community, lots of support. Facilitates the development of complex websites easily. Allows for fast development. Highly scalable. Build in security. Generates portions of the project for you. Good for tight deadlines. Ready to use “out of the box.” Abstracts the low-level parts of the system with an easy-to-use API. Supports extensions. Feature-rich. Internalization. Simple to use. Facilitates time zones well. The ORM is great, even for standalone applications, provides automatic schema migration. Allows for middleware. Jinja2 can be plugged in for flexibility and speed. More powerful and robust than Flask. Web socket support. Easy to set up URL routing.

**Cons:** Not intuitive. Requires doing things “the Django way.” Daunting to start development. So huge it might be hard to start using. Comes with things you’ll never need that slows it down. Easily confusing. Template system isn’t powerful. ORM support of automatic schema migrations has issues. Imperfect integration with graph databases. Unorganized tutorials. Necessary to use a third party library for configuring. Difficult to implement certain features, like live data binding. Hard to upgrade. Longer development cycle than Ruby on Rails.

**Grails**

**Pros:** Can use any existing Java library, metaprogramming, can use DSL, easy to convert between file types. Easy to start. Don’t have to edit a lot of XML. Very similar to Java. Dynamic. MVC pattern. Easy to process web data with Groovy, allows dynamic typing. Easy to manipulate and fetch data from database. Intuitive to use. Platform-independent. Rapid development. Good for small and medium size projects. Good documentation. Easy to manage the CSS for front end websites.

**Cons:** Groovy—no experience. Plugins are subpar. Hard to work in-depth with. The build and plugin mechanisms aren’t great. Small community. Not a lot of problem-solving forums out there, lots of problems have to be solved yourself. Small plugin ecosystem. Current version very unstable. Testing sucks. Run time language. Hard to maintain and read. Interpreted languages add weight at run time. Not easy to integrate.

**Laravel**

**Pros:** Supports namespaces. Supports middleware. Follows MVC framework. Basic CRUD functionality. Very organized, many development tools. Easy to add plug-ins (like Bootstrap). Promotes clean design and fast application development. Component-based. Event-oriented. CRUD is time-saving. Extensive documentation. Stable framework with authentication. SQL drivers. Automatic package discovery. Responsive interface.

**Cons:** Uses Yii. Cores are written in Laravel namespace and not everything uses the namespace slash (confusing). Complex to set up. Can be difficult to change. Updates quickly and difficult to migrate.

**Meteor**

**Pros:** Rapid development. Really easy to use. “Basically plug and play.” Very fast. Works very well with MongoDB. Helpful community. Speed. Good to develop smaller, reactive applications. Full stack. Mix of front-end and back-end JavaScript, as well as HTML, CSS, etc. Highly reactive UIs that auto-refresh. Integrates with Cordova, AngularJS, React, MongoDB. Authorization. Strong ecosystem, good documentation. Real time updates. Can build the front end, back end, and database with just JavaScript. Smart packages. Can structure projects however you want. Very modular.

**Cons:** Only uses MongoDB—no experience. Weak documentation. Very frequent updates that don’t always work together, unstable. Hard to debug. Error call stack is complicated. May not be great for large-scale applications. Less flexible. No Meteor-based UI components. Steep learning curve. Potential scaling problems. No native widget library. Reliant on strong network connection at all times.

**Ruby on Rails**

**Pros:** Good for prototyping quickly. Well documented, big community. Plugins and tutorials abound; most things have already been done for you by someone else. Easy to start. Provides everything you need to start. Fast and useful updates. Framework settings are flexible. Automated tests are easy to build. Great database compatibility. Quick deployment. “Convention over configuration.” “Allows small teams to do big things.” High productivity. High readability. High maintainability. Allows new developers to become productive quickly. Features can be implemented quickly. Works for just about any purpose. Easy to do test-driven development (TDD).

**Cons: “**Terrible for scaling,” not meant for dynamic projects. Takes a long time. Steep learning curve; Ruby—no experience. Doesn’t run well on platforms other than Linux (may not be a problem if we write and run it on Linux). Hard to master. If you have to write something yourself, it’s complex and time-consuming. Not good for projects that may grow. Hard to update. Very slow, especially as the app grows; sluggish framework. Compromises ease of use for performance. Long stack. Lot of work to integrate with other frameworks (such as AngularJS). Uses lots of RAM, hard to make real time applications with it. Exceptions become a problem for HUGE applications.

**Spring**

**Pros:** Simple to use, easy to set up, highly configurable. Components are overridable, easy to set up. Fast start for new projects. Easy and quick deployment. Quick development time. Dependency injections helps decouple code. Can use XML to configure different parts for different clients. Well-maintained. Bugs are fixed quickly. Updates to newer versions are easy. Junit tests are easy to implement. Is easy to configure the same base code for multiple clients with different client specific implementations. MVC framework. Highly reusable code, small in length. Integration with other frameworks is easy. Modularity. Flexible for configuration. Easy to test. Java—we have experience in that.

**Cons:** Security is a challenge; pieces can be overwritten so easily it’s hard to know what needs to be turned on or configured a specific way. A lot of XML becomes difficult to manage. XML data isn’t really proper validation, can create bugs. Can be difficult and time consuming. Compile time errors are common. Requires some high level configuration. Complex. Has a high learning curve. Lack of guidelines.