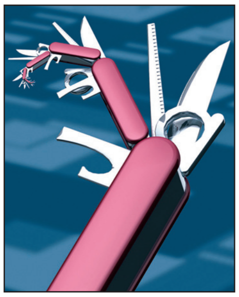

The Snap Framework

A Web Toolkit for Haskell

Gregory Collins • *Google Switzerland*
Doug Beardsley • *Karamaan Group*

Haskell is an advanced functional programming language. The product of more than 20 years of research, it enables rapid development of robust, concise, and fast software. Haskell supports integration with other languages and has loads of built-in concurrency, parallelism primitives, and rich libraries. With its state-of-the-art testing tools and an active community, Haskell makes it easier to produce flexible, maintainable, high-quality software. The most popular Haskell implementation is the Glasgow Haskell Compiler (GHC), a high-performance optimizing native-code compiler.

Here, we look at Snap, a Web-development framework for Haskell. Snap combines many other Web-development environments' best features: writing Web code in an expressive high-level language, a rapid development cycle, fast performance for native code, and easy deployment in production.

Why Is Haskell Good for Web Programming?

Haskell lets you write elegant, high-level code that rivals the performance of lower-level, imperative languages. You can write declarative programs at a high level of abstraction and expressiveness, while still maintaining excellent performance. When you need to do bare-metal bit-twiddling or need access to a C library that doesn't yet have a Haskell equivalent, its foreign function interface lets you easily drop down to C.

Writing solid, real-world code is easier in Haskell than in other languages. It has strong static typing, so many common programming errors, such as null pointer exceptions, can't occur. You can separate an application's core logic from the parts that must interact with the outside world: "pure" functions in Haskell, given the same inputs, always produce the same output. This property means that you almost always decompose a Haskell program into smaller constituent parts that you can test independently. Haskell's ecosystem also includes many powerful testing and code-coverage tools.

Haskell also comes out of the box with a set of easy-to-use primitives for parallel and concurrent programming and for performance profiling and tuning. Applications built with GHC enjoy solid multicore performance and can handle hundreds of thousands of concurrent network connections. We've been delighted to find that Haskell really shines for Web programming.

What's Snap?

Snap offers programmers a simple, expressive Web programming interface at roughly the same level of abstraction as Java servlets. It includes a fast, built-in HTTP server that drives application logic, so you can quickly create high-performance Web applications. Unlike some other Web frameworks in higher-level languages, Snap lets you consume request data and stream response data using a constant amount of memory.

Snap features excellent documentation and tutorial materials, and, for a young project, it's quite robust. Haskell's testing tools let you easily write a test suite with a high level of coverage. Snap uses Haskell's cross-platform libraries to run on all major operating systems. A Snap-based application also supports rapid development — an app running in development mode will load changes to your source code as soon as you make them. When you put your code into production, you can create a single, fast, standalone executable that's easy to deploy.

We designed Snap for efficiency; where appropriate, we use fast system calls such as `sendfile()` and `epoll()`. On older versions of

GHC, we have an optional binding to the libev C library for high-scalability I/O event scheduling.

Getting Started with Snap

Snap requires the Haskell platform (http://hackage.haskell.org/platform), which ships with a binary installer for most major platforms. The Haskell platform includes the GHC compiler, documentation, libraries, and tools; think of it as "Haskell: batteries included."

Once you've installed the Haskell platform, you can download, build, and install Snap's framework and all its dependencies using the cabal package manager:

```
$ cabal update
$ cabal install snap
```

The Snap installation builds the snap executable, which you can use to get started with a basic Snap project. By default, cabal installs executables to $HOME/.cabal/bin; the following instructions assume that this directory is on your $PATH.

To set up a new example Snap project (which is intended for use as an example or jumping-off point for your own Snap Web applications), run these commands:

```
$ mkdir myproject
$ cd myproject
$ snap init
```

The snap init command creates a skeletal Snap project in the current directory. This is a complete working Snap application that doesn't do much but that you can modify to start fleshing out your first project.

The snap init command creates the Main module for this project in src/Main.hs. When you build this project with cabal install, an executable called myproject is created in $HOME/.cabal/bin. To build and run the example application, execute these shell commands:

```
$ cabal install
$ myproject -p 8000
```

Now, point your Web browser to http://localhost:8000/; the server should respond with the default canned response.

Programming with Snap: A Simple Example

Let's look more closely at what's in the default skeleton application. (This article describes a prerelease version of Snap 0.3, which is scheduled to be released before this issue goes to print. Some details might change slightly.)

The snap init command creates a few files in the src/ subdirectory, the important ones being Application.hs, which contains the application's state definitions, and Site.hs, which contains your site's Web handlers.

First, let's look at some code from Application.hs:

```
type Application = SnapExtend
ApplicationState
```

This line indicates that your application extends the Snap type (the type of basic Web handlers without any associated in-memory state) with the ApplicationState type, which we'll define next (see Figure 1).

```
data ApplicationState = ApplicationState
   { templateState :: HeistState Application
   , timerState :: TimerState
   }
```

Figure 1. The ApplicationState *data type contains in-memory application state, which persists between requests.*

Snap applications hold some state in memory that they use to service requests. Things in this category include templates and caches. In our particular case, ApplicationState contains the set of templates we use to service user requests (using our heist templating library), as well as a TimerState, an example extension that stores the last time the application was reloaded.

Application.hs also includes code to initialize the application on load or reload. In Snap, we call this code an Initializer, which handles a Snap application's startups, reloads, and cleanup (see Figure 2). This code loads all the template files from the resources/templates directory, initializes the reload timer, and creates our ApplicationState object.

```
applicationInitializer :: Initializer ApplicationState
applicationInitializer = do
   heist <- heistInitializer "resources/templates"
emptyTemplateState
   timer <- timerInitializer
   return $ ApplicationState heist timer
```

Figure 2. An Initializer *handles a Snap application's startup, reloads, and cleanup.*

Let's turn our attention to src/Site.hs, which contains our application's Web handler code. The skeleton application contains two example handlers. One answers requests to get the site's root page; the other answers requests to /echo/foo by printing a message containing the foo input string (see Figures 3 through 5).

You can read this as, "If we're

```
index = ifTop $ heistLocal (bindSplices indexSplices) $
render "index"
 where
   indexSplices =
   [ ("start-time," startTimeSplice)
   , ("current-time," currentTimeSplice)
   ]
```

Figure 3. The root or homepage handler. This handler binds a couple of `heist` *template splices and renders the index template.*

```
...
<table id="info">
 <tr>
   <td>Config generated at:</td>
   <td><start-time/></td>
 </tr>
 <tr>
   <td>Page generated at:</td>
   <td><current-time/></td>
 </tr>
</table>
...
```

Figure 4. An example of splices in `heist`*. The* `start-time/` *and* `current-time/` *tags are bound to Haskell code, which renders HTML output.*

```
echo = do
   message <- decodedParam "stuff"
   heistLocal (bindString "message" message) $ render "echo"
 where
   decodedParam p = fromMaybe <*> urlDecode <$> fromMaybe ""
<$> getParam p
```

Figure 5. The `/echo` *handler. This code answers a request to* `/echo/foo` *by printing a message containing the string* `foo`*.*

at the root page, bind the following HTML tags and render the index template." A splice, in `heist` parlance, is a procedure you can bind to an XHTML tag that produces markup that gets spliced into the HTML output. Here, we're saying that the `<start-time/>` tag should report the time the server was last restarted and that the `<current-time/>` splice should report the current Web server time. Looking at `resources/templates/index.tpl`, you can see these splices at work (see Figure 4).

The handler in Figure 5 picks the `stuff` parameter out of the request's parameter map (we'll explain where that comes from in a second), URL-decodes it, binds the resulting message to a splice, and renders the echo template.

Finally, we can look at the main handler entry point (see Figure 6). This sets up a routing table for the site's URLs. Requests for the / URL are routed to the `index` handler; requests for `/echo/foo` are routed to the `echo` handler after we set `stuff=foo` in the request's parameter map.

The site handler's second half, after the main routing table, serves any files from the disk. The `a <|> b` syntax means "try `a`; if it fails, try `b`." In this case, if the user requests a URL that doesn't match any of the routing-table entries — for example, `/screen.css` — the `fileServe` function tries to find the file under the `resources/static` directory and serves it back to the user if it's found. If `fileServe` can't find the file, the `fileServe` handler fails, causing the `site` handler to fail, which causes Snap to produce a "404 Not Found" response.

Benchmarks

We compared Snap to five common Web frameworks: Ruby on Rails 2.3.5 (using the internal server), Grails 1.2.2, Apache 2.2.16, PHP 5.2.14, and Node.js 0.2.4.

Our comparison involved two benchmarks, which we ran on a quad-core Xeon machine running at 2.33 GHz. The pong benchmark (see Figure 7a) is basically "Hello, World!" It responds to requests by sending the string `pong`. (For the Apache/PHP line in this figure, we used Apache to serve the file and Apache with `mod_php` to issue the pong response.) The file benchmark (see Figure 7b) measures how fast each server can send a single 49-Kbyte image file. We obtained the benchmark numbers by using the popular `httperf` benchmarking tool.

Snap performed fairly well. With logging turned on, it served files roughly 40 percent faster than Apache; with logging turned off, it was more than 2.5 times as fast.

For more information about our testing methodology, including links to the source code, visit http://snapframework.com/benchmarks.

What's in Store for Snap?

Haskell is a great language that tends to inspire a quasireligious devotion among its adherents. This isn't necessarily because Haskell programmers are dogmatic or driven by concerns of ideological purity. It's because the qualitative experience of hacking in Haskell, once you've learned it, induces a Zen-like calm that few other languages come close to approaching. Compared to traditional imperative languages, Haskell often feels like slipping into a warm bath. Once you've gotten your program past the type checker and QuickCheck has tested your code against thousands of randomly generated test cases, you can feel comfortable that your program works correctly, with a lot less anxiety, sweat, and tears.

The flip side is that Haskell's learning curve is fairly steep. Programmers coming from traditional languages must unlearn many programming habits that serve them well in other languages. This is because Haskell is fundamentally different — even basic things, such as how values are computed from expressions, are different and unfamiliar. However, Haskell's popularity has increased exponentially in recent years, and we're hopeful that more programmers will take the time to check out a language that offers many tangible advantages.

Snap is a young library early in its evolutionary life cycle. The basic core is solid, and we plan to add features such as file upload support, HTML5 parsing, user authentication, and facilities for building larger Web applications out of smaller, pluggable parts. This is largely a question of time and manpower; we're confident we'll be able to offer a complete set of Web-programming facilities. However, up to now, we've concentrated on building a stable, efficient substrate upon which to build higher-level features.

```
site = route [ ("/," index)
             , ("/echo/:stuff," echo)
             ]
       <|> fileServe "resources/static"
```

Figure 6. The top-level site handler. It handles a couple of URL endpoints (/ and `/echo`) and shows an example of chaining together Snap handlers with the <|> operator.

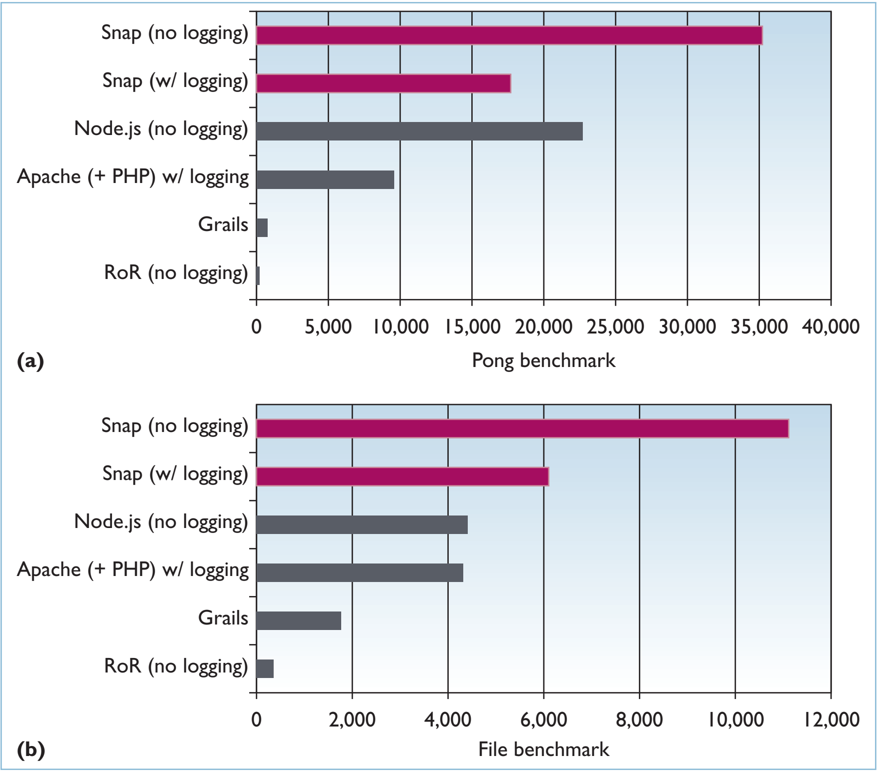

Figure 7. Comparing Snap with other common Web frameworks using the (a) pong and (b) file benchmarks; values in requests per second (higher is better.)

Snap is seeing some early adoption. Right now, we believe it's most interesting to Web developers who care about performance and expressiveness and who enjoy rolling up their sleeves and learning about exciting new techniques to build fast, solid Web applications. We welcome contributions of bug reports, requests for improvement, and especially code. If you'd like to contribute or experiment with Snap, more information, tutorials, and documentation are available at http://snapframework.com.

Gregory Collins is a site reliability engineer at Google Switzerland. Contact him at greg@gregorycollins.net.

Doug Beardsley is a quantitative developer at the Karamaan Group. Contact him at mightybyte@gmail.com.