Building Web Applications with Erlang

Zachary Kessin
Building Web Applications with Erlang

Why choose Erlang for web applications? Discover the answer hands-on by building a simple web service with this book. If you're an experienced web developer who knows basic Erlang, you'll learn how to work with REST, dynamic content, web sockets, and concurrency through several examples. In the process, you'll see first-hand that Erlang is ideal for building business-critical services.

Erlang was designed for fault-tolerant, non-stop telecom systems, and building applications with it requires a large set of skills. By the end of the book, you'll have the information you need to build a basic web service and get it running.

- Explore the power of Erlang and REST for building web services
- Serve static and dynamic content with the Yaws web server
- Use different methods for outputting data to the user, such as encoding Erlang data structures into JSON or XML
- Build an application to listen for HTTP requests, process them, store data, and return useful data
- Go beyond the request-response model—push data to clients with web sockets
- Use Erlang and Yaws to stream data from the server to a client

“A book which is truly needed and will help get Erlang to the next level.”

—Francesco Cesarini
Founder and Technical Director of Erlang Solutions, co-author of Erlang Programming

Purchase the ebook edition of this O'Reilly title at oreilly.com and get free updates for the life of the edition. Our ebooks are optimized for several electronic formats, including PDF, EPUB, Mobi, and DAISY—all DRM-free.
Building Web Applications with Erlang

Zachary Kessin
# Table of Contents

**Preface** .......................................................................................................................... vii

1. **Building Scalable Systems with Erlang and REST** .................................................. 1  
   Why Erlang? 1  
   Erlang’s Advantages 2  
   Lack of Types 3  
   OTP—For More Than Just Telecom! 4  
   Why Web Services? Why REST? 4  
   New Opportunities for Scaling and Resilience 6  
   Cloud Computing 6  
   System Architecture and Erlang Scaling 7  
   Data Storage Options 9

2. **Getting Started with Yaws** ......................................................................................... 15  
   Working with Yaws 16  
   Starting Yaws 16  
   Serving Static Files 17  
   Compiling, Loading, and Running Code 18  
   Clustering Yaws 20  
   Dynamic Content in Yaws 21  
   EHTML 24  
   Headers and Redirects 25  
   Templates 26  
   ErlyDTL 26  
   Logging 30  
   Erlang OTP error_logger 31

3. **Appmods: Dynamic Content in Yaws** ................................................................. 33  
   Appmod Configuration 34  
   When the URI Does Not Correspond to a File 34  
   Cookies 35
Session Handling 36
Access Control 38
Interacting with Erlang Services and Business Logic Layers 39

4. Implementing REST .................................................. 41
   Decoding a Request 41
     Extracting the User’s Request 41
   Response and Headers 43
   Building the Response 45
     JSON 47
     XML 49
   Responding to the REST Request 51
   A Full Example 51

5. File Upload ........................................................... 59
   The File Upload Request 59
   Saving to Disk 61
   Putting It All Together 63
   Storage in a Distributed System 65
   Saving to Amazon S3 66

6. WebSockets .......................................................... 69
   The WebSocket Request 70
   Basic WebSocket Handler 72
   Advanced WebSocket Handler 74

7. Streaming ............................................................ 79
   Simple Streaming 79

8. Using the HTTP Client .............................................. 83
   Making a Request 83
   Using OAuth 86
     Facebook Canvas 86

9. Building an Application with OTP ............................... 91
   Directory Structure 92
   Building an Application Server 93
   The Generic Server 93
   The Multicast Server 96
   Interfacing the Server with the Web 101
   Some Client-Side Code 102
   Let’s Have Some Real Supervision And... Here! 104
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Little Optimization</td>
<td>108</td>
</tr>
<tr>
<td>Bundling as an Application</td>
<td>114</td>
</tr>
<tr>
<td>The App File</td>
<td>115</td>
</tr>
<tr>
<td>Wrapping Up OTP</td>
<td>117</td>
</tr>
<tr>
<td>A. Installing Erlang and Yaws</td>
<td>119</td>
</tr>
<tr>
<td>B. Beyond Yaws</td>
<td>121</td>
</tr>
<tr>
<td>C. Interfacing with Ruby and Python</td>
<td>125</td>
</tr>
<tr>
<td>D. Using Erlang with Emacs</td>
<td>129</td>
</tr>
</tbody>
</table>
Erlang promises to let you build robust, fault-tolerant servers far more easily than with Java or C#. It almost sounds too good to be true, but Erlang has become a programmer’s secret handshake. As much as many of us hate our phone company, there is a basic truth that must be recognized: when you pick up your phone to make a call, it normally just works. So people have started to realize that telecom folks must be doing something right!

Erlang was built to program telephone switches at Ericsson, and most of the language design choices reflect what was necessary to program a telephone switch. That means, for example, that Erlang software can run for years at a time without interruption because phone switches are expected to do that. Erlang applications can be upgraded in place without taking the system offline or even losing state because the phone company can’t drop a city’s worth of calls every time they have to patch a bug or roll out a new feature.

When a web service goes down, a lot of things break. It may not be as obvious as a suddenly interrupted call, but it may actually create more problems as failures create new failures. Web services can benefit from the language design decisions Erlang’s creators made in a telephone switching environment. Having a server that can run without interruption can allow a development team to provide a better service to their customers.

**Who This Book Is For**

This book shows you the baby steps to building a web service with Erlang. It does not try to teach you Erlang (there are other books for that), nor does it try to show you how to build the large-scale applications that really call for Erlang. Instead, it shows you how to build simple web services as a step along the way to learning to build large-scale web services.

I expect that many readers will, like me, be long-time web professionals who are looking at Erlang as a way to stand out from a crowd of Java and C# developers. After all, in a few years Erlang may be the next big thing, and you want to be ahead of the wave. Or
perhaps you have become frustrated with some aspect of building web applications in those other languages and are looking for something a bit more powerful.

You need to know at least basic Erlang, but you should also be familiar with web development—in PHP, Perl, Ruby, Java, or something else. I assume that you have seen HTML and know the basics of how HTTP works.

There are a few examples in this book that use JavaScript to interface a browser with the Erlang example. Except in Chapter 9, this code is not critical to understanding what the Erlang code is doing, although of course if you are building a large web application it will contain JavaScript. I also use CoffeeScript in a few places. CoffeeScript is a small language that compiles down to JavaScript and generally makes for a much nicer programming experience than straight JavaScript.¹

## Learning Erlang

This book will not teach you Erlang. There are already a number of good resources for that, including:

- **Erlang Programming**, by Francesco Cesarini and Simon Thompson, published by O’Reilly.
- **Programming Erlang**, by Joe Armstrong, published by The Pragmatic Programmers.

Reading the first few chapters of any of these and understanding the basics of how Erlang works should be enough. However, you should plan to really work through those chapters and write some simple programs before attempting the projects here.

In particular, you should read up on sequential code and the very basics of how concurrency works in Erlang. When building large-scale applications in Erlang, taking advantage of the Open Telecom Platform (OTP) will allow the programmer to leverage a large amount of well-tested functionality. And while OTP is very powerful and will make development in Erlang much easier, the details of OTP are less important to learn up front and can be learned as you go along after you have an understanding of how other parts of the system work.

## Before You Start

Before you dive into this book, you should have Erlang and Yaws installed on your system. (If you need help in this, check Appendix A.) Erlang and Yaws can be run on Windows, Mac, and Linux, so any type of system will work fine.

¹. You can find more information about CoffeeScript at [http://coffeescript.org](http://coffeescript.org).
Several people have asked me why I wrote this book around Yaws and not some other web package. There were a few reasons. First of all, Yaws seemed the easiest package to get something simple working in. Second, several of the other packages do not support web sockets (or at least didn’t when I started writing), and I knew that I would be needing web sockets in my own development.

I am also assuming that you are familiar with the Unix command line. While it is not necessary to be a Bash Kung-Fu Master (I’m not), you should be able to interact with the bash shell and not freak out.

**What You Will Learn**

Building a full Erlang application requires a large set of skills. This book will help you get to the point where you can build a basic web service application and get it running. First, you’ll explore some of the power and mystery of Erlang and REST. You’ll see why Erlang makes sense as a foundation for building scalable and reliable systems and why REST is a popular approach to building web services and explore some of the tradeoffs involved in using the two together. This first chapter will also explore some of your data storage options.

The Yaws web server is the foundation of our application, so you’ll learn to configure Yaws and serve static content. Yes, static content. In many cases, a website with dynamic content will have a collection of static files as resources. Once you know how to manage static files, you can move on to working with dynamic content, embedding Erlang into an HTML file or other kind of file (see “Dynamic Content in Yaws” on page 21). You’ll learn about working with HTTP itself and basic debugging tools like logging.

You’ll need a way to route client requests presented as URLs to the internal resources of your service. Appmods, discussed in Chapter 3, will let you map arbitrary URLs onto relevant resources.

Next we cover output formats. I will show three general ways to output data to the user. The first, and least useful, method is to use ehtml to directly translate Erlang data into HTML or XML. We also will see how to use the erlydtl library to use the Django template language to create formatted output. (DTL is a common template package on Python and should be familiar to some readers of this book.) Finally, we will see how to encode Erlang data structures into JSON or XML, which can be sent to the user. In many cases, modern web applications will have a page of static (or almost static) HTML and a lot of JavaScript that will interact with the server by sending JSON or XML over Ajax channels.

Now that we can generate output, it’s time to build a simple RESTful service. You’ll assemble an application that can listen for HTTP requests, process them, store data,
and return useful information. You’ll also learn how to handle large chunks of incoming information, dealing with multipart requests and file uploads.

If you’d like to go beyond HTTP’s request-response model, Chapter 6 presents a live bidirectional method of communication between the client and the server. Yaws supports web sockets, and the dynamic, event-driven nature of Erlang makes for an ideal platform for pushing dynamic data to the client.

Finally, Chapter 9 presents a somewhat larger example that pulls together most or all of the previously discussed topics into one complete application. This chapter will show how to build a complete small application with Yaws and OTP.

### The Limits of This Book

If you want a complete guide to building large, fault-tolerant sites with Erlang, you’ll be disappointed. The architecture of a large-scale website requires a book of its own. (A project like that will probably end up being 90% backend and logic and 10% web interface.)

I also deliberately did not cover any of the half dozen or so frameworks for building web applications with Erlang, as I wanted to focus on the task of building a basic service in Erlang with just Yaws and custom code. MochiWeb, Chicago Boss, Nitrogen, Zotonic, and the rest need their own books, but I summarize them briefly in Appendix B.

This book does not attempt to show how to structure an Erlang application beyond the very basics: a full introduction to OTP requires a longer book than this one.

It is also not an introduction to supervision trees. They are covered briefly in Chapter 9, but this is a short introduction to a very large topic.

Erlang has a full set of features to allow it to monitor the state of an application and respond when processes or nodes go offline. This is amazingly powerful on many levels. For example, in the case of a node failing at 2:00 AM, Erlang can generate a log message and create a new node from a cloud with no need for human intervention—a far better scenario than an emergency wake up call for the sysadmin!

For automated testing, Erlang has a test framework called EUnit (documented in *Erlang Programming*) as well as a version of the Haskell QuickCheck testing suite. These are beyond the scope of this book, but can be quite useful for development.

Finally, this book does not cover details of how best to run Erlang on Amazon EC2 or other cloud services. Running a bunch of Erlang nodes on cloud hosts can make a lot of sense.
Help! It Doesn’t Compile or Run!

When working with a new framework in a language you may not know very well, it is inevitable that sooner or later you will hit a few problems. Code won’t compile, or else it will compile and then crash in all sorts of strange ways.

If you are anything like me, you probably won’t be doing a copy/paste of code directly from this book (though you are welcome to do so if you want); instead, you’ll probably try to adapt this code to some other problem you are trying to solve. After all, that’s the whole point of books like this—to give you tools to solve problems in fun new ways. So what should you do if something doesn’t work as expected?

Diagnosing the Error

If a request to Yaws does not work, it will show a screen link, as shown in Figure P-1. This may look a bit cryptic at first glance, but is actually quite helpful. First of all, you will notice the path to the file that contains the Erlang module with the offending code. Then you will see the reason why it crashed (in this case, a call to a function in an unloaded module), and then the request that was made and the stack trace. In Erlang R15 this stack trace will also include line numbers; this screen shot is from R14B02, which does not include them.

![Error Page](image)

**Figure P-1. Error Page**
What Version of Erlang and Yaws Are You Running?

This book was built around Erlang R14B02 and R15B. Ideally you should use R15B or later. This is a major release that among other features includes line numbers in stack traces, which makes finding errors much easier. You can find the version of Erlang you have by running `erl -v` from the command line.

This book was also built with Yaws version 1.92. You can find your version of Yaws by running `yaws -v` from the command line. The web sockets interface described in Chapter 6 changed in a major way between Yaws versions 1.90 and 1.92.

Is Everything Loaded Correctly?

Programmers who have come to Erlang from languages like PHP or Perl will find that there is an extra step in Erlang. While Yaws will automatically compile and load new `.yaws` files (see “Dynamic Content in Yaws” on page 21), any other Erlang module must be compiled and loaded into the Erlang runtime. Compilation can be done from within the Erlang shell by using the `c(Module)` command, which will also load the new code into the Erlang runtime. This is very useful for interactive testing of code and for the speed of your development cycle. It’s certainly possible that someone converting from PHP to Erlang will forget this step from time to time.

Erlang code can also be compiled from an external command line with the `erlc` command from a Unix shell. Erlang will autoload the code; however, it is important to set the include paths correctly so that it can find the `.beam` files. This option is good for doing things like automatic builds. The loading of external modules may be automated by adding the load commands to the `.erlang` file or other configuration options.

In addition, Erlang applications will often be composed of many modules, all of which must be loaded into the system for it to work. So if something fails, check to see if a module has not been loaded or is not in the path. To see the current path from the shell, run `code:get_path()`.

One nice thing about Erlang is that if the system is set up in a reasonable way, you should never need to take the entire system offline to upload a new version of code.

Are You Calling Everything Correctly?

The Erlang command line is your friend! This is a good place to try out your code and see if it works as expected. Don’t be afraid to create test data at the command line and give your functions test inputs to make sure that they return the correct results.

---

2. This also works with Cygwin on Windows
When you load a module, its records are not loaded into the shell. This has to be done explicitly with the `rr` command from the Erlang shell. You can also define a record with `rd` and remove a record with `rf`. To use these, type `help()` on the Erlang command line.

**Is Mnesia Running with Correct Tables?**

Mnesia, Erlang’s built-in database, has to be started up and tables created for it to work. Before you start Mnesia you have to run the command `mnesia:create_schema/1`, which creates the basic database storage for Mnesia; then, to start Mnesia use the command `application:start(mnesia)`. If you are having trouble with Mnesia tables, you can use the table viewer by typing `tv:start()` at the Erlang command prompt.

**Is the Example Just Plain Wrong?**

Obviously, I’ve tried to ensure that all the code in this book runs smoothly the first time, but it’s possible that an error crept through. You’ll want to check the errata on this book’s web page (see the How to Contact Us section at the end of the Preface), and download the sample code, which will be updated to fix any errors found after publication.

**Conventions Used in This Book**

The following typographical conventions are used in this book:

*Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

*Constant width*

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

*Constant width bold*

Shows commands or other text that should be typed literally by the user.

*Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.
About the Author

Zachary Kessin has been working on developing interactive web applications since 1994. In the last few years, Zachary’s focus has been on building complex applications in the browser with JavaScript, browser-based testing with Selenium, functional programming, and code generation.