THE Transparent Web

Functional, Reactive, Isomorphic

Christopher J. Wilson

MEAP

MANNING

MEAP Edition
Manning Early Access Program
The Transparent Web
Functional, Reactive, Isomorphic
Version 10

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for buying the MEAP for *The Transparent Web: Functional, Reactive, Isomorphic*. If the title isn't enough of a hint, this book is a little bit unlike many other tech books. It is a broad survey of what's on the horizon of web development.

This book requires a bit of programming experience. Whether you have created a handful of sites or you have many more under your belt, this is for you. You'll also need a dash of curiosity to get the most out of it. More concretely, I assume a little bit of object-oriented programming experience as well.

This MEAP starts with chapters 1, 2, and 5. Chapter 1 is a broad overview of all the concepts in the book. Chapter 2 dives into writing an application using an isomorphic web framework, Opa. And chapter 5 looks into what modern static typing brings to the table.

Over the course of the rest of the book, I'll dig into three big themes: functional, reactive, and isomorphic. Functional programming serves as the backdrop for the rest of the concepts in the book. Functional programming is less widespread in web application development, but it brings with it a rich toolbox of techniques for building complex apps. As it applies to web development, *isomorphic*, means being able to reuse the same code on both the client and the server. Looked at another way, I think of it as treating the client and server as one *unified platform.* More holistically, this also includes things like compiling native code to JavaScript, and creating applications which include their own operating system as a library. Lastly, we'll look at the concept of reactive programming. This flips the normal control flow of user-facing applications on its head. Instead of writing programs which expect to be in control, reactive programming handles interaction with the user in terms of what the user is doing. Surprisingly, the result isn't chaos, but clean, elegant apps. I look at reactivity in both JavaScript and Elm, a whole language built around these ideas.

This book has been a big undertaking for me. Now, I hope you'll join me in the MEAP process to make the book better. Let me know what you think, what could be changed, or what you'd like to see in later chapters. I look forward to seeing your feedback in the Author Online forum.

I can't thank you enough for taking time to pick up this book. I hope you'll find it useful, and thought-provoking.

— Chris Wilson

brief contents

1 Advancing The Web

Web development can often feel like writing the same thing twice. We first write database schemas and application logic for the server. Then on the client we have to implement much of the same logic in order to validate inputs and provide realtime feedback. We are able to share data but not the code that implements application logic.

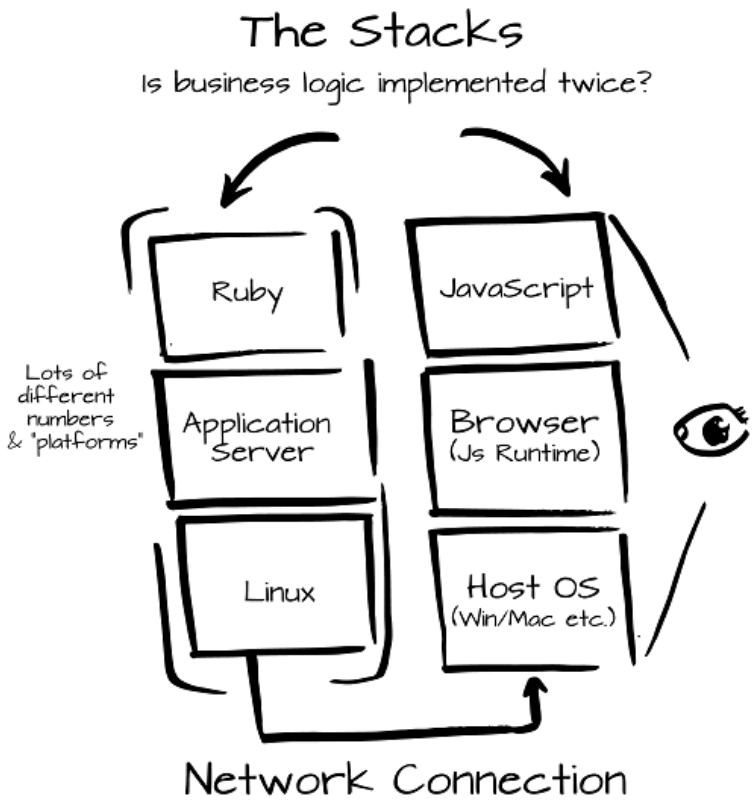
Even the way that we share that data seems awkward. We create routes structured around manipulating resources via a few distinct verbs: 'CREATE', 'READ', 'UPDATE', and 'DELETE'. This leaves us with a one size fits all API that has much more to do with how the web works than how our application is structured.

When we write user interfaces on the web, we do so by first downloading a document interspersed with formatting and structuring commands. We then write JavaScript to imperatively modify this document, swapping in new chunks here, or altering the display of existing parts there. Again, this whole process marches to the drummer of the way that the web works. HTML, derived from SGML, is a document markup language, not a user-interface system.

And when it comes to interactivity, JavaScript is what you get. Much in the way that Henry Ford quipped about the Model T, 'Any customer can have a car painted any color that he wants as long as it is black', JavaScript is the only language supported by all modern browsers. The cause for JavaScript's immense popularity eventually circles back around to the fact that it is massively popular. Whatever the merits of JavaScript, and it does have many, the fact that there's no choice is a drawback. These issues are present in current web development:

- Having to write similar code for multiple platforms (browser and server)
- Awkward or tedious client-server communication
- An ill-fitting UI language,
- An imperative and limited scripting language

Figure 1.1. Applications can end up being written twice



This book, 'The Functional Web', is ultimately about making the web a better place for developers to inhabit and, by extension, a better web for all of us to use. While we explore the landscape of web development we'll keep our eyes on a few important themes. These tie together the different chapters and concepts that make up this book. I think they serve as a valuable perspective on the changing landscape of the web.

The first theme I call 'unified stacks', an example of this is combining server-side and client-side code into one code base. Then there's 'functional programming', a particular style of writing code. Functional programming is a big topic. My interest with it in this book is how it informs writing clear, succinct code that expresses programmer intent. Static typing is also a theme that I will lump in with functional

programming throughout this book. Though it is neither a necessary nor sufficient condition for functional programming, static typing nonetheless fits well there. I feel that static typing is complementary to both functional programming and web development. It provides the structure while functional programming brings the dynamism. Lastly, there is the theme of 'reactive programming'. Like functional programming this is a big topic! As it applies to this book, it describes methods for orienting applications to be responsive to outside input. In a user interface, this means reacting to clicks and key presses. It is the viewpoint that an application is not a big run loop, but small functions run in response to outside events.

This book's mode is comparative and exploratory rather than prescriptive — I want to unearth options rather than try and find the "one true way." And because of that, it may come across as an odd tech book. Rather than an exhaustive tutorial of some technology, I'm going to introduce you to many things. This is for two reasons. Each chapter could easily be an entire book in its own right. Unfortunately, I have to be very succinct and skip a lot of richness in each topic. The second is that I want this to be an 'overview'. I want us to see enough different things that we can start to see how they resemble one another — not in their detail but in their broad strokes. We'll learn just enough about each new language or technology to see how it could fit as a future direction for programming.

It's worth pausing for a moment to emphasize that last statement. These are technologies which 'could' fit as a future direction. I do not have a crystal ball and so I'm making guesses, trying to spot trends. In reality I'm not sure if any of the technologies in this book will be the next big thing, that's way too hard to predict. The overwelming liklihood is that these won't be the next killer app in development. But what I do think is likely is that some or all of these technologies will be influential in whatever does emerge. As I'm writing this book, I'm seeing more and more cross-pollination between the JS/ES world and Elm — just to name one example. Back when I started, it was not at all clear that that would happen. But now it seems like 'The Elm Architecture' and 'React + Flux' are influencing one another in a difficult to tease apart way.

Many of the ideas herein draw from or touch on functional programming. But I don't consider this to be a book 'about' functional programming, rather it is a book about coming to grips with the complexity of modern web application development. It just so happens that functional programming has a lot to say about cutting that knot of complexity.

1.1 Major Themes of The Functional Web

When I first noticed Opa and Elm, two languages that we'll talk about in later chapters, there seemed to be some thread connecting them. Though they were developed by different people, they clearly shared some characteristics. They both allow the developer to program in just one language. The solutions differ a bit, but

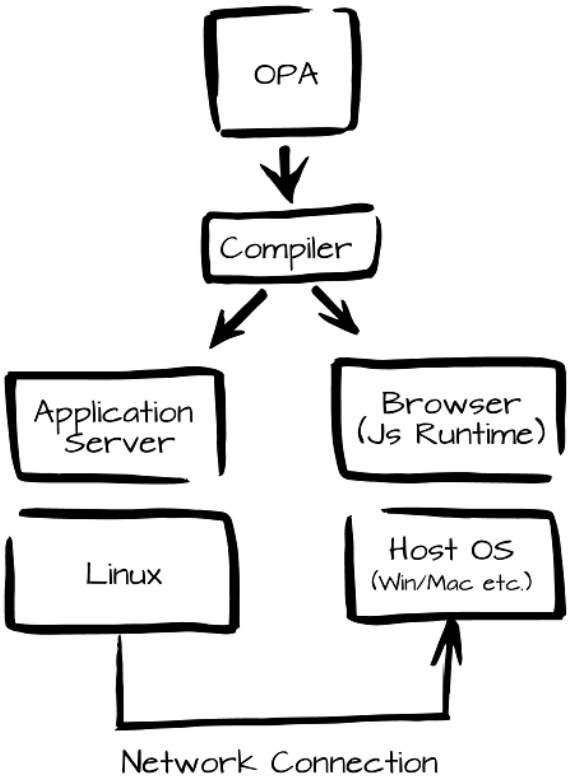
both of them allow you to write HTML without switching away from your language. They both include syntax for dealing with common web-related tasks (HTML, SQL, etc.). There were 'themes' underlying these separate languages.

Through this lens, I saw that there were many more technologies that seemed to fit. I realized that what Opa and Elm were really up to was simplifying, collapsing, or 'unifying' parts of the web stack. JavaScript, HTML, CSS and even, in Opa's case, the server-side could be programmed together. By bringing these differing languages together (and having a nice compiler) enables the language to check for problems. Errors like referencing a non-existent CSS class can be caught by the compiler.

In the following sections, I'll explain how each of these themes guide our exploration through the Functional Web.

1.1.1 Unified Stacks

Figure 1.2. Server and Client-side applications can be written in the same codebase



Web application programming has progressed to a much richer model. The big shift is that we now write an API server that consults a database and pair it with a

client-side application that makes requests of that API. This scheme might be called a "rich client-side application" or just "client-side application". In this model, the client-side application is often where most of the complexity resides. And there is complexity in the 'interplay' between the two sides as well. But this way of doing things has the benefit of being more flexible. The same backing-API may be used by many different front-ends be they web-based, iOS app, Android, and so on.

Combine client and server

The first example of the 'unifying platforms' approach to this architecture assumes the above as a given. We're going to have an app which is "split in half" and it communicates via an API. This is such a fundamental aspect of our client-server application, that the unified platform approach is to bring that communication code into the framework itself. In Opa, this detail is somewhat hidden by the compiler, but it is there. As a part of compiling our program, Opa determines the code needed to serialize data to and from the client — and indeed, what code should run on the server and what code should run on the client. This allows us to write code without regard for where it runs. There are still some necessary divisions, say for hiding sensitive business logic from the client-side, but these are divisions that are meaningful in the context of the application and are not just technological limitations.

Let's look at this client/server division in Opa. In most cases code will be available on both the client and the server. But there are times when it is important that code should 'definitely' be running only on the client or only on the server. Opa has a declaration syntax to cover that situation.

Listing 1.1. Where code should run in Opa

```
function client_or_server(x, y) { ... }        (1)

client function client_function(x, y) { ... }  (2)

server function server_function(x, y) { ... }  (3)
```

(1) Opa decides where to compile this function
(2) This function is compiled for the client-side
(3) This function is compiled for the server-side

The compiler is able to determine that certain things, such as database access must be compiled into server-side code, whereas reading the value of a CSS selector pertains to the client.

Combine the app and the OS

There are other ways that we'll explore this 'unifying' theme: unifying the server.

MirageOS is an example of a so-called 'library OS' or a 'unikernel'. The network, storage, and other drivers that would usually be a part of the operating system are instead compiled into a standalone program. This creates an executable that can run directly within a hypervisor (in this case, the Xen). Put another way, there is no Linux or Windows OS underlying the application!

This leads to applications which are 'extremely' lightweight. Our example application in Chapter 3, weighs in at about 4.5 MB — application and OS together. These applications can also boot very quickly, typically within 400 ms or so. Put together these properties have interesting implications for microservice architectures. Services can be offline and only booted when a DNS request arrives. By the time that a client resolves the DNS and makes a TCP request to the service, it'll have already booted. This scheme is implemented in the jitsu project (Just-In-Time Summoning of Unikernels).

1.1.2 Functional Programming

A whole book could be written about functional programming and many have been. To give you a brief definition, a functional language is one where you program with mathematical functions. Functions, in the mathematical sense, define a relationship between inputs and outputs. In programming, the meaning of 'function' is much broader than it is in mathematics. Functions as used in programming are often really procedures, lists of steps to be carried out, possibly with inputs and outputs. In the mathematical world, a function without inputs or outputs wouldn't make sense. There would be no way to convey information into or out of the function.

The goal of functional programming is to create more tractable code. When the only things that can affect the outcome of a function call are its arguments, it's much easier to find problems. There's a cool way of visualizing this, with credit to link:http://blog.jenkster.com/2015/12/what-is-functional-programming.html Kris Jenkins. Every function you write has 'two' sets of inputs and outputs. The first set of input/output is the functions' arguments and return value. The second kind of input is the preconditons and external state it depends on, and the second kind output is the changes it makes to the program state. The second set of input/output is 'hidden' or at least not as obvious, but it has as much of an effect as arguments which are passed in and the value it returns.

Reducing statefulness

'State' is the particular condition that a program is in at a particular time; it was like 'this' a minute ago, and now it's like 'that'. Functional programming demands a fundamental change in how you write software by erasing any notion of time or change. This means that each time a function is called it behaves the same way it did the last time. If the same arguments are provided to it, then it'll return the same answer. We can't tell the difference between the first time that we call a function

and the 100th. And at that point, it doesn' make any sense to make a distinction. There's no hidden record that the function is keeping, no history that it maintains anywhere.

Languages that work in this way are much easier to deal with, even if it doesn't seem so at first. Most tests can be written as simple input/output pairs. We save a lot of "party planning" in our programs because we're not worrying about setting up a particular configuration of objects 'just so'.

Composition and modularity

Software, if it's to have any hope of growing to a large size and still be practicable, must be written in a modular way out of composable pieces. In a well-designed system say you have two operations you'd like to perform, one after the other. It should be possible to combine those operations together to yield a third operation that is the combination of the two. Think of how arithmetic works for the basic operations of addition, subtraction and multiplication.

Let's say we have two different operations: adding two to a number and multiplying a number by 3. This can be expressed as two functions, f and g.

```
f(x) = x + 2
g(x) = x * 3
```

We can apply these functions successively. First we add two to a number and then we multiply it by 3. Let's call this new function, the combination of f and g, 'h'. I can write this new function that combines the two.

```
h(x) = g(f(x))
h(x) = g(x + 2)
h(x) = (x + 2) * 3
```

In each step I expanded out what the function was, its definition. The equals signs are significant because everything is equal to one another. At the end, we've ended up with a another function. Compositionality is being able to combine expressions like this, as deeply as we like. At the end we'll always end up with another function.

Expanding on the math example, here's a JavaScript function that uses a few other functions.

```
function foo1(input) {
    var x, y, z;

    x = bar(input);
    y = baz(x);
    z = quux(y);

    return z;
}
```

Now imagine that we didn't really care about what `foo`, `bar`, `baz`, or `quux` actually are. We're just focusing on the notion of calling a function, getting its result and then using that in a successive function. The function `foo1` is acting merely as a coordinator and we just want this skeleton. And since we're generalizing, we can't say exactly what functions should be called, so we'd better pass those in as arguments:

```
function foo2(input, f, g, h) {
    var x, y, z;

    x = f(input); y = g(x); z = h(y);

    return z;
}
```

And furthermore all the skeleton is really doing is plumbing of those functions together. We can more clearly express this:

```
function foo3(input, f, g, h) {
    return h(g(f(input)));
}
```

The true nature of foo is revealed! And given that simpler nature, we could express `foo` just as a result of other operations:

Listing 1.2. Final version of `foo`, written as a composition of functions

```
function compose(f, g) {
  return function(x) { return f(g(x)); };
}

function chain(funcs) {
  if (funcs.length === 1) {
    return funcs[0];
  }
  return compose(chain(funcs.slice(1)), funcs[0]); (1)
}

var foo4 = chain([bar, baz, quux]);
```

(1) `funcs.slice(1)` returns all elements of the `funcs` array after the first one. This operation is sometimes called "tail" or "rest".

I've showed we can take simpler parts, in this case the bar, baz, and quux functions and combine them to create more complex behavior.

Now I know that the idea that this is simpler may seem really far-fetched but there's a way in which it absolutely is. Once we've written `compose` and `chain`, which are handy in their own right, they allow us to see that `foo` is only gluing

other things together. We write `chain` once and put it in a library, or find a library that's already written it. The value we get is being able to write functions in a style like that of `foo4`. TThis expresses the essence of compositionality; we can understand code by what its pieces do and how they're combined. I think this is worth boiling down into a kind of motto:

The essense of functional programming

> *If you combine small, stateless pieces, that you understand individually, in standardized ways, the large programs you build are easier to understand.*

Functional programming seeks, as much as possible, to build programs by putting together such simple pieces like this.

Functional programming has lots of ideas worth borrowing. If you're new to the ideas of functional programming but familiar with the web, then I'm happy to say that this is a great place to be. Just like a tour guide, I'll be pointing out ideas that are coming from functional programming as we go along. There will be sections and chapters devoted to the ideas behind the technology that I'm discussing and not just the technology itself.

If you're familiar with concepts that I listed above, DSLs, type systems, reactive programming, well then, I'm hoping that you'll still find lots to learn as you see how these ideas can be applied to the web.

1.1.3 Reactive Programming

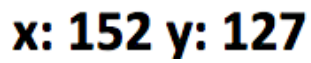
Reactive programming means structing our applications around how we'd like to respond to events from the outside world. We'll see a lot more about how this works when we look at Elm in Chapter 8, . In particular, we'll zero in on the variety of reactive programming that Elm espouses. Like functional programming itself, reactive programming is a broad style, and encompasses many other ideas.

```
main : Program Never Model Mouse.Position
main =
    Html.program
        { init = ( Model { x = 0, y = 0 }, none )
        , update = update
        , subscriptions = mousePositionSubscription
        , view = view
        }


mousePositionSubscription : Model -> Sub Mouse.Position
mousePositionSubscription _ =
    let
        mouseMoveFunction pos =
            pos
    in
        Mouse.moves mouseMoveFunction
```

This produces outputs like, `x: 22, y: 345`, in the uppermost left corner of the window.

Figure 1.3. A screenshot showing the current mouse position



`Mouse.moves` is a higer-order function that takes a function as its argument. The passed-in function is supplied with the current mouse position, here that role is handled by `mouseMoveFunction`. `mouseMoveFunction` returns whatever argument is passed to it, in this case that's a mouse position, `pos`. Astute readers may note that this is the 'identity function' (a function which returns whatever its argument is). The overall type of `mousePositionSubscription` is `Model → Sub Mouse.Position`, but we are ignoring the `Model` argument, that's what the underscore, `_`, is saying.

A subscription in Elm is exactly what it sounds like. The program is able to register its interest in some kind of event. In the example above, that's the position of the mouse.

1.2 Summary

Web development is in a transitional period. Since starting modestly in the mid '90s, rich web applications have become a key part of modern computing. We're seeing a Cambrian explosion in web application development. Web development has a lot to borrow from the world of functional programming and I believe ideas like those explored in this book, collectively or in part, represent the future of programming.

In this chapter, we saw that:

- I believe the web is in a transitional stage. Parts of the web stack will coalesce while others will absorb ideas from functional programming. This will make programming for the web more coherent.
- "The Functional Web" consists of the themes of 'Unified stacks', 'functional programming', and 'reactive programming'.
- We delved into each of these themes to get a sense for how they fit.

This book proceeds through a series of examples and discussions that will demonstrate what these concepts are and how they can be applied. Where possible, I'll also point out analogous features between the different frameworks and techniques. This will make the new ideas clearer by showing many of them in a few different ways.

If all that sounds good then let's get started!