Kweb: The Easier Way to Website

Ian Clarke

Apr 20, 2019

Contents

Create beautiful, efficient, and powerful websites in Kotlin, quickly.

This documentation is a work-in-progress, feedback is very welcome. Please submit an issue, or fork, improve, and create a pull request to contribute directly.

And please join us to discuss all-things Kweb in our Gitter channel, where we'll be happy to help if you have questions.

CHAPTER 1

Introduction

1.1 Motivation

Most websites consist of at least two tightly coupled components. One runs in the browser, and the other runs on a web server.

- **Client**
  - Runs in the browser
  - Responsible for user interaction
  - Typically written in JavaScript
  - Untrusted execution environment
  - Unreliable persistent local state
- **Server**
  - Runs in a datacenter
  - Responsible for business logic
  - May be written in a wide variety of languages
  - Trusted execution environment
  - Reliable persistent global state

The fact that these two tightly coupled components are often written in different languages and must communicate with each other over a HTTP connection adds significant complexity to the overall system.

This is the problem Kweb was designed to solve. We do this by moving as much of the logic to the server as possible, leaving a simple but powerful interface to the web browser where server-browser communications are handled automatically.

Kweb includes a typesafe domain-specific language for building and manipulating the DOM in a remote web browser.

Kweb runs on Kotlin, an excellent modern programming language that is rapidly growing in popularity. Kotlin is now Google's recommended language for Android development. According to Github, Kotlin was the fastest growing programming language of 2018.

1.2 How does it work?

Kweb is a self-contained Kotlin library that can be added easily to new or existing projects. When Kweb receives a HTTP request it responds with a small HTML file including optimized instructions for building the page, and a client which connects back to the web server via a WebSocket. The client then waits and listens for instructions from the server.

To minimize latency, Kweb can preload instructions to the browser to modify the DOM instantly in response to browser events, perhaps to disable a button or temporarily display a "spinner". This allows Kweb to respond instantly to user behavior without a server round-trip.

Kweb is designed to be efficient. All operations are handled asynchronously, thread and memory usage are minimized. Kweb runs on the JVM, which is 5-10 times faster than Node.js.

Kweb also takes an end-to-end approach to state. You can bind the value of a DOM element to a field in your database, and have it update in realtime automatically.

1.3 Features

- Free as in speech, licenced under LGPL v3.0
- A unified codebase for your webapp, from database to DOM
- End-to-end Kotlin (Why Kotlin?)
- Bind DOM values directly to a value in your database and have them update in realtime
- Statically typed HTML DSL - let your IDE catch bugs so you don't have to
- Efficient server-side rendering, DOM caching, and instruction preloading
- Very lightweight, Kweb is less than 5,000 lines of code

1.4 Relevant Links

- Website: https://kweb.io/
- Source code: https://github.com/kwebio/core
- Demo: http://demo.kweb.io:7659/
- Feedback: https://github.com/kwebio/core/issues
- Help: https://gitter.im/kwebio/Lobby
- API Docs: https://jitpack.io/com/github/kwebio/core/0.3.15/javadoc/

CHAPTER 2

Getting Started

2.1 What you'll need

Some familiarity with Kotlin is assumed, as is familiarity with Gradle. You should also have some familiarity with HTML.

2.2 Adding Kweb to your project

Kweb is distributed via JitPack, so add this to the repositories {block} in your build.gradle:

```
repositories {
  maven { url 'https://jitpack.io' }
}
```

Then add Kweb to the dependencies block:

```
dependencies {
  compile 'com.github.kwebio:core:LATEST_VERSION'
}
```

Replace LATEST_VERSION with the latest version of Kweb, which you can find on JitPack.

2.3 Hello world

Create a new Kotlin file and type this:

```
import io.kweb.*
import io.kweb.dom.element.*
import io.kweb.dom.element.creation.tags.*
```

(continues on next page)

(continued from previous page)

```
fun main() {
  Kweb(port = 16097) {
    doc.body.new {
      h1().text("Hello World!")
    }
 }
}
```

Run it, and then visit http://localhost:16097/ in your web browser to see the traditional greeting, translating to the following HTML body:

```
<body>
  <h1>Hello World!</h1>
</body>
```

This simple example already illustrates some important features of Kweb:

- Getting a kwebsite up and running is a breeze, no messing around with servlets, or third party webservers
- Your Kweb code will loosely mirror the structure of the HTML it generates

2.4 Hello world[2]

One way to think of Kweb is as a domain-specific language (DSL) for building and manipulating a DOM in a remote web browser.

One of the great things about Kotlin is its ability to embed DSLs, so you can use the full power of Kotlin, including control flow features such as *for loops*:

Here is a simple example using a *for* loop:

```
import io.kweb.*
import io.kweb.dom.element.*
import io.kweb.dom.element.creation.tags.*

fun main() {
  Kweb(port = 16097) {
    doc.body.new {
      ul().new {
          for (x in 1..5) {
             li().text("Hello World $x!")
          }
      }
    }
 }
}
```

To produce...

```
<body>
  <ul>
      <li>Hello World 1!</li>
      <li>Hello World 2!</li>
      <li>Hello World 3!</li>
```

(continues on next page)

(continued from previous page)

```
      <li>Hello World 4!</li>
      <li>Hello World 5!</li>
  <ul>
</body>
```

CHAPTER 3

DOM Basics

3.1 Creating DOM Elements

The DOM is built starting with an element, typically the BodyElement which is obtained easily as follows:

```
import io.kweb.*
import io.kweb.dom.element.*

fun main() {
  Kweb(port = 16097) {
     val body : BodyElement = doc.body
 }
}
```

Let's create a button element as a child of the body element, we do this using the *.new* function (which is supported by all Element types):

```
import io.kweb.*
import io.kweb.dom.element.*

fun main() {
  Kweb(port = 16097) {
     doc.body.new {
         val button = button().text("Click Me!")
     }
   }
 }
```

As you can see, it's easy to set the text of an element, you can also modify its attributes:

```
button.setAttribute("class", "bigbutton")
```

Or delete it:

```
button.delete()
```

3.2 Reading from the DOM

Kweb can also read from the DOM, in this case the value of an <input> element:

```
import io.kweb.Kweb
import io.kweb.dom.element.creation.tags.*
import io.kweb.dom.element.events.on
import io.kweb.dom.element.new
import io.kweb.state.KVar
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.future.await
import kotlinx.coroutines.launch

fun main() {
    Kweb(port = 2395) {
        doc.body.new {
            val input: InputElement = input()
            input.on.submit {
                GlobalScope.launch {
                    val value = input.getValue().await()
                    println("Value: $value")
                }
            }
        }
    }
}
```

Note that input.getValue() returns a CompletableFuture<String>. This is because it can take up to several hundred milliseconds to retrieve from the browser, and we don't want the application to block if it can be avoided. Here we use Kotlin's very powerful coroutines features to avoid any unnecessary blocking.

Note: We discuss a better way to do this in a subsequent section.

3.3 Supported HTML tags

Kweb supports a significant subset of HTML tags like *button()*, *p()*, *a()*, *table()*, and so on. You can find a more complete list in the API documentation (scroll down to the *Functions* section). This provides a nice statically-typed HTML DSL, fully integrated with the Kotlin language.

If a tag doesn't have explicit support in Kweb that's not a problem. For example, here is how you might use the infamous and now-obsolete <blink> tag:

```
doc.body.new {
    val blink = element("blink").text("I am annoying!")
}
```

3.4 Extending Kweb to support new HTML tags

Adding support for new tags to Kweb is easy, take a look at the source. If you add some useful functionality please submit a pull request via Github, or just ask us and we'll do our best to add support.

3.5 Further Reading

The Element class provides many other useful ways to interact with DOM elements.

CHAPTER 4

Event Handling

4.1 Listening for events

You can attach event handlers to DOM elements:

```
doc.body.new {
    val label = h1()
    label.text("Click Me")
    label.on.click {
        label.text("Clicked!")
    }
}
```

Most if not all JavaScript event types are supported, and you can read event data like which key was pressed:

```
doc.body.new {
    val input = input(type = text)
    input.on.keypress { keypressEvent ->
        println("Key Pressed: ${keypressEvent.key}")
    }
}
```

4.2 Immediate events

Since the code to respond to events runs on the server, there may be a short lag between the action causing the event and any changes to the DOM caused by the event handler. This was a common complaint about previous server-driven web frameworks like Vaadin, inhibiting their adoption.

Fortunately, Kweb has a solution - onImmediate:

```
doc.body.new {
    val label = h1()
    label.text("Click Me")
    label.onImmediate.click {
        label.text("Clicked!")
    }
}
```

This is identical to the first event listener example, except *on* has been replaced by *onImmediate*.

Kweb executes this event handler *on page render* and records the changes it makes to the DOM. It then "pre-loads" these instructions to the browser such that they are executed immediately when the event occurs without any server round-trip.

Warning: Due to this pre-loading mechanism, the event handler for an *onImmediate* must limit itself to simple DOM modifications. Kweb includes some runtime safeguards against this but they can't catch every problem so please use with caution.

4.3 Combination event handlers

A common pattern is to use both types of event handler on a DOM element. The immediate handler might disable a clicked button, or temporarily display some form of spinner. The normal handler would then do what it needs on the server, and then perhaps re-enable the button and remove the spinner.

CHAPTER 5

State Management

5.1 Building Blocks

Kweb makes use of the observer pattern, through the KVar class.

A KVar contains a single typed object, which can change over time. For example:

```
val counter = KVar(0)
```

Here we create a counter of type *KVar<Int>* initialized with the value 0.

We can also read and modify the value of a KVar:

```
println("Counter value ${counter.value}")
counter.value = 1
println("Counter value ${counter.value}")
```

Will print:

```
Counter value 0
counter value 1
```

KVars support powerful mapping semantics to create new KVars:

```
val counterDoubled = counter.map { it * 2 }
counter.value = 5
println("counter: ${counter.value}, doubled: ${counterDoubled.value}")
counter.value = 6
println("counter: ${counter.value}, doubled: ${counterDoubled.value}")
```

Will print:

```
counter: 5, doubled: 10
counter: 6, doubled: 12
```

Note that counterDoubled updates automatically.

Note: KVars should only be used to store values that are themselves immutable, such as an Int, String, or a Kotlin data class with immutable parameters.

5.2 KVars and the DOM

You can use a KVar (or KVal) to set the text of a DOM element:

```
val name = KVar("John")
li().text(name)
```

The neat part is that if the value of *name* changes, the DOM element text will update automatically. It may help to think of this as a way of "unwrapping" a KVar.

Numerous other functions on Elements support KVars in a similar manner, including innerHtml() and setAttribute().

5.3 Binding a KVar to an <input> element's value

For <INPUT> elements you can set the value to a KVar, note that this connection is bidirectional, so any changes to the KVar will be reflected in realtime in the browser, and similarly any changes in the browser by the user will be reflected immediately in the KVar:

```
Kweb(port = 2395) {
    doc.body.new {
         p().text("What is your name?")
        val clickMe = input(type = text)
        val nameKVar = KVar("Peter Pan")
        clickMe.value = nameKVar
        p().text(nameKVar.map { n -> "Hi $n!" })
    }
}
```

5.4 Rendering state to a DOM fragment

But what if you want to do more than just modify a single element based on a KVar, what if you want to modify a whole tree of elements?

This is where the render function comes in:

```
val list = KVar(listOf("one", "two", "three"))

Kweb(port = 16097) {
    doc.body.new {
        render(list) { rList ->
            ul().new {
                for (item in rList) {
                    li().text(item)
                }
```

(continues on next page)

(continued from previous page)

```
                }
            }
        }
}
```

Here, if we were to change the list:

```
list.value = listOf("four", "five", "six")
```

Then the relevant part of the DOM will be redrawn instantly.

The simplicity of this mechanism may disguise how powerful it is, since render {} blocks can be nested, it's possible to be very selective about what parts of the DOM must be modified in response to changes in state.

Note: Kweb will only re-render a DOM fragment if the value of the KVar actually changes. Because of this it is most efficient to avoid "unwrapping" KVars with a *render()* or *.text()* call before you need to. The KVal.map {} function is a powerful tool for manipulating KVals and KVars without unwrapping them.

5.5 Reversible mappings

If you check the type of *counterDoubled*, you'll notice that it's a *KVal* rather than a *KVar*. KVal's values may not be modified directly, so this won't be permitted:

```
counterDoubled.value = 20 // <--- This won't compile
```

The *KVar* class has a second map() function which takes a *ReversableFunction* implementation. This version of *map* will produce a KVar which can be modified, as follows:

```
val counterDoubled = counter.map(object : ReversableFunction<Int, Int>("doubledCounter
↪") {
    override fun invoke(from: Int) = from * 2
    override fun reverse(original: Int, change: Int) = change / 2
})
counter.value = 5
println("counter: ${counter.value}, doubled: ${counterDoubled.value}")

counterDoubled.value = 12 // <--- This wouldn't have worked before
println("counter: ${counter.value}, doubled: ${counterDoubled.value}")
```

Will print:

```
counter: 5, doubled: 10
counter: 6, doubled: 12
```

CHAPTER 6

URL Routing

In a web application, routing is the process of using URLs to drive the user interface (UI). URLs are a prominent feature in every web browser, and have several main functions:

- Bookmarking - Users can bookmark URLs in their web browser to save content they want to come back to later.
- Sharing - Users can share content with others by sending a link to a certain page.
- Navigation - URLs are used to drive the web browser's back/forward functions.

Traditionally, visiting a different URL witin the same website would cause a new page to be downloaded from the server, but current state-of-the-art websites are able to modify the page in response to URL changes without a full refresh.

With Kweb's routing mechanism you get this automatically.

6.1 A simple example

```
import io.kweb.Kweb
import io.kweb.dom.element.new
import io.kweb.dom.element.creation.tags.h1
import io.kweb.routing.route

fun main() {
    Kweb(port = 16097) {
        doc.body.new {
            route {
                path("/users/{userId}") { params ->
                    val userId = params.getValue("userId")
                    h1().text(userId.map { "User id: $it" })
                }
                path("/lists/{listId}") { params ->
                    val listId = params.getValue("listId")
                    h1().text(listId.map { "List id: $it" })
```

(continues on next page)

(continued from previous page)

```
                }
            }
        }
    }
}
```

Now, if you visit http://localhost:16097/users/997, you will see:

```
<h1>User id: 997</h1>
```

You can have as many path()s as you need, each with it's own path definition. The definition can contain parameters wrapped in {braces}.

The value of these parameters can then be retrieved from the *params* map, but note that the values are wrapped in a *KVar<String>* object. This means that you can use all of Kweb's state management features to render parts of the DOM using this value.

The key advantage here is that if the URL changes the page can be updated without a full page refresh, but rather only changing the parts of the DOM that need to change - this is much faster and more efficient.

6.2 Handing 404s

You can override the default 404 Page Not Found message in the event that none of the routes match, making it easy to integrate the 404 page with the style of your overall website:

```
route {
    path("/users/{userId}") { params ->
        // ...
    }
    notFound {
      h1().text("Page not found!")
    }
}
```

6.3 Modifying the URL

You can obtain *and modify* the URL of the current page using *url(simpleUrlParser)* within the Kweb {block}. This returns a KVar<URL > which you can use to read and *modify* the page URL:

```
import io.kweb.Kweb
import io.kweb.dom.element.creation.tags.a
import io.kweb.dom.element.new
import io.kweb.routing.route
import io.kweb.state.*

fun main() {
    Kweb(port = 16097) {
        doc.body.new {
            val path = url(simpleUrlParser).path
            route {
                path("/") {
                    path.value = "/number/1"
```

(continues on next page)

(continued from previous page)

```
                }
                path("/number/{num}") { params ->
                    val num = params.getValue("num").toInt()
                    a().text(num.map {"Number $it"}).on.click {
                        path.value = "/number/${num.value + 1}"
                    }
                }
            }
        }
    }
}
```

If you visit http://localhost:16097/ the URL will immediately update to http://localhost:16097/number/1 without a page refresh, and you'll see a hyperlink with text "Number 1". If you click on this link you'll see that the number increments (both in the URL and in the link text), also without a page refresh.

An even more elegant approach that would also work would be to replace:

```
path.value = "/number/${num.value + 1}"
```

...with...

```
num.value++
```

This would have the exact same effect because the KVars always work bidirectionally, so can be used both to read and modify that part of the page URL, resulting in an automatic re-render of the necessary DOM elements.

CHAPTER 7

Database

7.1 Overview

Kweb integrates nicely with Shoebox, a key-value store that supports the observer pattern, and a sister project to Kweb. Shoebox has both in-memory and persistent (filesystem) engines.

In the future Shoebox will support back-end cloud services like AWS Pub/Sub Messaging and Dynamo DB, which would enable unlimited scalability.

We'll assume you've taken a minute or two to review Shoebox and get the general idea of how it's used.

7.2 Shoebox and State

This example shows how *toVar* can be used to convert a value in a Shoebox to a KVar:

```
fun main() {
    data class User(val name : String, val email : String)
    val users = Shoebox<User>()
    users["aaa"] = User("Ian", "ian@ian.ian")

    Kweb(port = 16097) {
        doc.body.new {
            val user = toVar(users, "aaa")
            ul().new {
                li().text(user.map {"Name: ${it.name}"})
                li().text(user.map {"Email: ${it.email}"})

            }
        }
    }
}
```

7.3 Working Example

For a more complete example of using Shoebox for persistent storage see the to do demo.

CHAPTER 8

Aesthetics

Kweb has out-of-the-box support for the excellent Fomantic UI framework, which helps create beautiful, responsive layouts using human-friendly HTML.

Kweb's Fomantic UI plugin provides a convenient DSL to use Fomantic UI.

8.1 Getting started

First tell Kweb to use the fomantic UI plugin:

```
import io.kweb.plugins.fomanticUI.*

fun main() {
    Kweb(port = 16097, plugins = listOf(fomanticUIPlugin)) {
        // ...
    }
}
```

Now the plugin will add the Fomantic UI CSS and JavaScript code to your website automatically.

Now, let's look at one of the simple examples from the Fomantic UI documentation:

```
<div class="ui icon input">
  <input type="text" placeholder="Search...">
  <i class="search icon"></i>
</div>
```

Translates to the Kotlin:

```
import io.kweb.plugins.fomanticUI.*
import io.kweb.dom.element.creation.tags.InputType.*

fun main() {
    Kweb(port = 16097, plugins = listOf(fomanticUIPlugin)) {
```

(continues on next page)

(continued from previous page)

```
        div(fomantic.ui.icon.input).new {
            input(type = text, placeholder = "Search...")
            i(fomantic.search.icon)
        }
    }
}
```

8.2 Other UI Frameworks

It's easy to create Kweb plugins for many JavaScript tools and frameworks, taking full advantage of Kotlin's DSL capabilities.

The Fomantic UI plugin implementation itself can serve as an example.

8.3 Example and Demo

See a simple app built using Fomantic UI and Kweb (with source): http://demo.kweb.io:7659/

CHAPTER 9

Frequently Asked Questions

9.1 Won't Kweb be slow relative to client-side web frameworks?

No, Kweb's immediate events allow you to avoid any server communication delay by responding immediately to DOM-modifying events. This should address the majority of scenarios where a server-driven approach might otherwise be sluggish.

9.2 What's the difference between Kweb and Vaadin?

Of all web frameworks we're aware of, Vaadin is the closest in design and philosophy to Kweb. In many ways Kweb is a philosophical descendant of Vaadin. This makes Vaadin one of the most useful frameworks to compare Kweb to, as there are also very important differences:

- Kweb is *far* more lightweight than Vaadin. At the time of writing, kwebio/core is about 4,351 lines of code, while vaadin/framework is currently 502,398 lines of code, almost a 100:1 ratio!
- Vaadin doesn't have any equivalent feature to Kweb's immediate events, which has led to frequent complaints of sluggishness from Vaadin users because a server round-trip is required to update the DOM.
- Vaadin brought a more desktop-style of user interface to the web browser, but since then we've realized that users generally prefer their websites to look like websites.
- This is why Kweb's philosophy is to be a thin interface between server logic and the user's browser, leveraging existing tools from the JavaScript ecosystem when it makes sense.
- Kweb was built natively for Kotlin, and takes advantage of all of its language features like coroutines and the flexible DSL-like syntax. Because of this Kweb code can be a lot more concise, without sacrificing readability.
- In Vaadin's favor, it has been a commercial product since 2006, it is extremely mature and has a vast developer ecosystem, while Kweb is still pre-1.0.

9.3 Is there a larger working example?

Yes, here is a simple todo list implementation which demonstrates many of Kweb's features.

You can find a copy of this demo running here: http://demo.kweb.io:7659/

It's running on a $50/month EC2 instance. Try visiting the same list URL in two different browser windows and notice how they synchronize in realtime.

9.4 What about templates?

Kweb replaces templates with something better - a typesafe HTML DSL embedded within a powerful programming language.

If you like you could separate out the code that interfaces directly to the DOM - this would be architecturally closer to a template-based approach, but we view it as a feature that this paradigm isn't forced on the programmer.

9.5 How is "Kweb" pronounced?

One syllable, like "queue" and "web" smashed together.

9.6 I have a question not answered here

Please join us in our Gitter chat room where we'll do our best to answer any questions you might have.