Kweb: The Easier Way to Website

Ian Clarke

Jan 19, 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

Coders work hard to try to make the website appear like a single piece of software to the website visitor, but this requires a lot of hidden complexity.

Kweb lets you write your entire app as a single piece of software. It does this by moving as much of the logic to the server as possible, leaving a simple but powerful interface to the web browser. This allows you to build beautiful responsive website much more quickly.

It's written in Kotlin, a modern powerful programming language that is rapidly growing in popularity. For example, Kotlin is now being promoted by Google as a replacement for Java in Android development.

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 solves one of the most serious problems affecting some of Kweb's philosophical ancestors, Wicket and Vaadin (which shared Kweb's "server driven" approach).

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
- Connect to almost any back-end key/value store and pub/sub service for unlimited scalability

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'
}
```

You can find the LATEST_VERSION of Kweb on JitPack.

2.3 Hello world

Create a new Kotlin file and type this:

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

```

(continues on next page)

(continued from previous page)

```
fun main() {
  Kweb(port = 8091) {
    doc.body.new {
      h1().text("Hello World!")
    }
 }
}
```

Visit http://localhost:8091/ in your web browser and you should 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]

We have the full expressiveness of Kotlin at our disposal. Witness the power of the 'for' loop:

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

fun main() {
  Kweb(port = 8091) {
    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>
      <li>Hello World 4!</li>
      <li>Hello World 5!</li>
  <ul>
</body>
```

CHAPTER 3

DOM Basics

3.1 Modifying the DOM

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 = 8091) {
     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):

```
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 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 an tag doesn't have explicit support in Kweb that's not a problem. For example, here is how you might use the famous <blink> tag:

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

3.3 Extending Kweb to support new HTML tags

Adding support for new tags to Kweb is fairly simple. You can see how the existing functions are implemented. Feel free to submit a pull request via Github, or just submit an issue and we'll do our best to add support.

3.4 Reading the DOM

You can read values from the DOM too:

```
doc.body.new {
    val label = h1().text("What is your name?")
    val clickMe = input(type = text)
    clickMe.setValue("Foo Bar")
    GlobalScope.launch {
        val v : String = clickMe.getValue().await()
        label.text("Value: $v")
    }
}
```

Notice that *clickMe.getValue()* doesn't return a String, it returns a CompletableFuture<String>. This is because retrieving something from the DOM requires some communication with the browser and will take some time - and we don't want to block while we wait.

This allows us to take advantage of Kotlin's coroutines functionality to make this fairly seamless to the programmer (using GlobalScope.launch and await).

Yes, this example is a little pointless since we're just setting the value and then immediately reading it, more realistic use cases will follow.

3.5 Next steps

Kweb really comes into its own when the above is combined with Kweb's approach to State Management, particularly the render { } function.

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 server-driven web frameworks like Vaadin, inhibiting their adoption.

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 can contain a value of any type, 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 how counterDoubled updates automatically.

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.

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

5.3 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 = 1234) {
    doc.body.new {
        render(list) { rList ->
            ul().new {
                for (item in rList) {
                    li().text(item)
                }
            }
        }
    }
}
```

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

---

5.4 Routing with KVars

You can obtain *and modify* the URL of the current page using *url(simpleUrlParser)*. This returns a KVar<URL >, which you can then use with *render* to handle however you wish.

Note: You can modify this KVar<URL> and the browser URL bar and DOM will update accordingly, but *without* a page refresh.

```
Kweb(port = 1234) {
    doc.body.new {
        val url: KVar<URL> = url(simpleUrlParser)
        render(url.path) { path ->
            ul().new {
                for (p in path) {
                    li().text(p)
                }
            }
        }
    }
}
```

And that's pretty-much all you need to know to handle URL routing in your app, although we will make more specific recommendations later.

5.5 KVars and Persistent Storage

While you don't have to use it, Kweb integrates nicely with Shoebox, a key-value store that supports the observer pattern. Shoebox has both in-memory and persistent (on disk) engines, and new engines can be added quite easily.

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

This example shows how *toVar* can be used to convert a value in a Shoebox to a KVar, and use it with the DOM as previously described:

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

    Kweb(port = 1234) {
        doc.body.new {
            val user = toVar(users, "aaa")
            ul().new {
                li().text(user.map {"Name: ${it.name}"})
                li().text(user.map {"Email: ${it.email}"})

            }
        }
    }
}
```

5.6 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

Aesthetics

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

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

6.1 Getting started

First tell Kweb to use the Semantic UI plugin:

```
import io.kweb.plugins.semanticUI.*

fun main() {
    Kweb(port = 4736, plugins = listOf(semanticUIPlugin)) {
        // ...
    }
}
```

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

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

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

We can translate this to Kweb fairly directly:

```
import io.kweb.plugins.semanticUI.*

fun main() {
    Kweb(port = 4736, plugins = listOf(semanticUIPlugin)) {
        div(semantic.ui.icon.input).new {
```

(continues on next page)

(continued from previous page)

```
            input(type = text, placeholder = "Search...")
            i(semantic.search.icon)
        }
    }
}
```

6.2 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 Semantic UI plugin implementation itself can serve as an example.

CHAPTER 7

Frequently Asked Questions

7.1 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.

7.2 What's the difference between Kweb and Vaadin?

Vaadin comes with it's own set of UI widgets which you have to use, whereas with Kweb you can use your favorite JavaScript UI framework (Semantic UI is supported out of the box). Also, being designed for Kotlin rather than Java allows Kweb to take full advantage of Kotlin's numerous benefits, including a much more ergonomic API.

7.3 Is there a larger working example?

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

- Website: http://kweb.io/
- Source code: https://github.com/kwebio/core
- 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/