
Kweb: The Easier Way to Website

Ian Clarke

Feb 15, 2019

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 1.1 | Motivation | 3 |
| 1.2 | How does it work? | 4 |
| 1.3 | Features | 4 |
| 1.4 | Relevant Links | 4 |
| 2 | Getting Started | 5 |
| 2.1 | What you'll need | 5 |
| 2.2 | Adding Kweb to your project | 5 |
| 2.3 | Hello world | 5 |
| 2.4 | Hello world ² | 6 |
| 3 | DOM Basics | 9 |
| 3.1 | Modifying the DOM | 9 |
| 3.2 | Supported HTML tags | 10 |
| 3.3 | Extending Kweb to support new HTML tags | 10 |
| 3.4 | Reading the DOM | 10 |
| 3.5 | Next steps | 10 |
| 4 | Event Handling | 11 |
| 4.1 | Listening for events | 11 |
| 4.2 | Immediate events | 11 |
| 4.3 | Combination event handlers | 12 |
| 5 | State Management | 13 |
| 5.1 | Building Blocks | 13 |
| 5.2 | KVars and the DOM | 14 |
| 5.3 | Rendering state to a DOM fragment | 14 |
| 5.4 | KVars and Persistent Storage | 15 |
| 5.5 | Reversible mappings | 15 |
| 6 | URL Routing | 17 |
| 6.1 | A simple example | 17 |
| 6.2 | Modifying the URL | 18 |
| 7 | Aesthetics | 21 |
| 7.1 | Getting started | 21 |

| | | |
|----------|--|-----------|
| 7.2 | Other UI Frameworks | 22 |
| 8 | Frequently Asked Questions | 23 |
| 8.1 | Won't Kweb be slow relative to client-side web frameworks? | 23 |
| 8.2 | What's the difference between Kweb and Vaadin? | 23 |
| 8.3 | Is there a larger working example? | 24 |
| 8.4 | I have a question not answered here | 24 |

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.

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](#), while also being popular for backend architectures (where it's often used as a drop-in replacement for Java).

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
- Ridiculously lightweight, Kweb is less than 5,000 lines of code

1.4 Relevant Links

- 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/>

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 = 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 world²

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.*

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

(continues on next page)

(continued from previous page)

```
<ul>  
</body>
```


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 = 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 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 infamous and now-obsolete `<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 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.

3.4 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.

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.

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 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)
                }
            }
        }
    }
}
```

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.

5.4 KVars and Persistent Storage

Kweb integrates nicely with [Shoobox](#), a key-value store that supports the observer pattern, and a sister project to Kweb. Shoobox has both in-memory and persistent (on disk) engines.

In the future Shoobox 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 [Shoobox](#) and get the general idea of how it's used.

This example shows how `toVar` can be used to convert a value in a Shoobox 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 = Shoobox<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}" })
            }
        }
    }
}
```

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 `ReversibleFunction` implementation. This version of `map` will produce a `KVar` which can be modified, as follows:

```
val counterDoubled = counter.map(object : ReversibleFunction<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
```


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 within 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.routing.route
import io.kweb.state.*

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)


```
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.

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.

7.1 Getting started

First tell Kweb to use the Semantic UI plugin:

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

fun main() {
    Kweb(port = 16097, 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 Kotlin fairly directly thanks to Kweb's HTML DSL:

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

fun main() {
    Kweb(port = 16097, 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)
    }
}
```

7.2 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.

Frequently Asked Questions

8.1 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.

8.2 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.

8.3 Is there a larger working example?

Yes, here is a simple [todo list](#) implementation which demonstrates many of Kweb's features.

8.4 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.