
Kweb: Pleasant web development, in Kotlin

Ian Clarke

Sep 02, 2020

Contents

1	Introduction	3
1.1	Motivation	3
1.2	How does it work?	3
1.3	Features	3
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
2.5	Template Repository	7
3	DOM Basics	9
3.1	Creating DOM Elements and Fragments	9
3.2	Reading from the DOM	10
3.3	Supported HTML tags	10
3.4	Further Reading	11
4	Event Handling	13
4.1	Listening for events	13
4.2	Immediate events	13
4.3	Combination event handlers	14
5	Observer Pattern & State	15
5.1	Overview	15
5.2	Building blocks	15
5.3	KVars and the DOM	16
5.4	Binding a KVar to an input element's value	16
5.5	Rendering state to a DOM fragment	17
5.6	Extracting data class properties	17
5.7	KVals & Reversible mapping	17
6	URL Routing	19
6.1	A simple example	19
6.2	Handling 404s	20
6.3	Modifying the URL	20

7	Database	23
7.1	Overview	23
7.2	Shoebox and State	23
7.3	Future Development	24
7.4	Other Databases	24
7.5	Working Example	24
8	CSS & Style	25
8.1	Getting started	25
8.2	Other UI Frameworks	26
8.3	Example and Demo	26
9	Frequently Asked Questions	27
9.1	Won't Kweb be slow relative to client-side web frameworks?	27
9.2	What's the difference between Kweb and Vaadin?	27
9.3	Is there a larger working example?	28
9.4	How do I enable HTTPS?	28
9.5	Can I embed Kweb within an Android app?	28
9.6	What about templates?	28
9.7	Why risk my project on a framework I just heard of?	28
9.8	How is "Kweb" pronounced?	28
9.9	Can Kweb be embedded in an existing Ktor app?	29
9.10	I have a question not answered here	29

A new way to create beautiful, efficient, and scalable websites in Kotlin, quickly.

If you have questions, suggestions, or if you encounter a problem then please [submit a Github issue](#), we respond to most questions within a few hours.

1.1 Motivation

Modern websites consist of at least two **tightly coupled** components, one runs in the browser, the other on the server. These are often written in different programming languages and must communicate with each other over an HTTP connection.

Kweb's goal is to eliminate this server/browser separation so that your webapp's architecture is determined by the problem you're solving, rather than the limitations of today's tools.

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 the initial HTML page, and some JavaScript that connects back to the web server via a WebSocket. The page then waits and listens for instructions from the server, while notifying the server of relevant browser events.

A common concern about this approach is that the user interface might feel sluggish if it is server driven. Kweb solves this problem by **preloading** instructions to the browser to be executed immediately on browser events without a server round-trip.

Kweb is built on the excellent **Ktor** framework, which handles HTTP, HTTPS, and WebSocket transport. You don't need to know Ktor to use Kweb, but if you've already got a Ktor app you can **embed Kweb as a Feature**.

1.3 Features

- Allows the problem to determine your architecture, not the server/browser divide
- End-to-end Kotlin (**Why Kotlin?**)
- Keep the web page in sync with your back-end data in realtime, Kweb does all the plumbing for you

- Server-side HTML rendering with [rehydration](#)
- Efficient instruction preloading to avoid unnecessary server communication
- 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/kweb-core>
- API documentation: <http://dokka.kweb.io/kweb-core/>
- Example projects: <https://github.com/kwebio/kweb-demos>
- Live Demo: <http://demo.kweb.io:7659/>
- Questions/Feedback/Bugs: <https://github.com/kwebio/kweb-core/issues>

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' }
    jcenter()
}
```

Then add Kweb to the dependencies block:

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

    // This (or another SLF4J binding) is required for Kweb to log errors
    compile group: 'org.slf4j', name: 'slf4j-simple', version: '1.7.30'
}
```

Replace LATEST_VERSION with the latest version of Kweb, which you can find on <https://jitpack.io/#kwebio/kweb-core>.

2.3 Hello world

Create a new Kotlin file and type this:

```
import kweb.*

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, while also listening for and handing DOM events.

Importantly, this DSL can also do anything Kotlin can do, including features like for loops, functions, coroutines, and classes.

Here is a simple example using an ordinary Kotlin *for loop*:

```
import kweb.*

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>
```

You can use functions for modularization and reuse:

```
fun main() {
    Kweb(port = 16097) {
        doc.body.new {
            ul().new {
                for (x in 1..5) {
                    createMessage(x)
                }
            }
        }
    }
}

private fun ElementCreator<ULElement>.createMessage(x: Int) {
    li().text("Hello World $x!")
}
```

As you can see this is an extension function, which allows you to use the Kweb DSL within the newly created function.

Don't worry if you're unsure about this because you can use IntelliJ's [extract function](#) refactoring to create these functions automatically.

2.5 Template Repository

You can find a simple template Kweb project in [kwebio/kweb-template](#).

3.1 Creating DOM Elements and Fragments

The DOM is built starting with an `Element`, normally the `BodyElement`, which is obtained easily as follows:

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

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

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

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

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

If you assign the button element to a `val` then you can also modify its attributes:

```
val button = button()
button.text("Click Me!")
button.classes("bigbutton")
button.setAttributeRaw("autofocus", true)
```

Or equivalently using Kotlin's `apply` scope function:

```
button().apply {
    text("Click Me!")
    classes("bigbutton")
    setAttributeRaw("autofocus", true)
}
```

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 kweb.*
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 an even better way to read `<input>` values in the [Observer Pattern & State](#) 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 `prelude.kt` (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 Further Reading

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

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:

```
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 Overview

Kweb uses the [observer pattern](#) to manage state.

A Kweb app can be viewed as a mapping function between state on the server and the DOM within the end-user's web browser. Once this mapping is defined, simply modify this state and the change will propagate automatically to the browser.

5.2 Building blocks

A `KVar` class 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}")
counter.value++
println("Counter value ${counter.value}")
```

Will print:

```
Counter value 0
Counter value 1
Counter value 2
```

`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.

5.3 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()`.

5.4 Binding a KVar to an input element’s value

For `<input>` elements you can set the value to a KVar, which will connect them bidirectionally.

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, for example:

```
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!" })
    }
}
```

This will also work for `<option>` and `<textarea>` elements which also have values.

See also: [ValueElement.value](#)

5.5 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 so you should 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.

5.6 Extracting data class properties

If your KVar contains a `data class` then you can use `Kvar.property()` to create a KVar from one of its properties which will update the original KVar if changed:

```
data class User(val name : String)
val user = KVar(User("Ian"))
val name = user.property(User::name)
name.value = "John"
println(user) // Will print: KVar(User(name = "John"))
```

5.7 KVals & Reversible mapping

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
```

Note: Reversible mappings are an advanced feature that you only need if you want the mapped value to be a mutable *KVar*. Most of the time the simple `KVal.map {}` function will be what you need.

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 kweb.Kweb
import kweb.dom.element.new
import kweb.dom.element.creation.tags.h1
import 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 its 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 `WebBrowser.url`.

This returns a `KVar<String>` which contains the URL relative to the origin - so for the page `http://foo/bar/z` the `url` would be `/bar/z`.

Here is a more realistic example:

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

fun main() {
    Kweb(port = 16097) {
        doc.body.new {
            route {
```

(continues on next page)

(continued from previous page)

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

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.

The line `num.value++` is worthy of additional attention as there is more going on here than meets the eye. `num` is a `KVar<Int>`, and so it can be incremented via its `value` property. This will cause the page URL to update, which will in-turn cause the DOM to update to reflect the new URL. All of this is handled for you automatically by Kweb.

7.1 Overview

Shoobox is a simple key-value store that supports the [observer pattern](#), and is a sister project to Kweb.

We'll assume you've taken a minute or two to review [Shoobox](#) and get the general idea of how it's used.

7.2 Shoobox and State

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

```
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}"})
            }
        }
    }
}
```

7.3 Future Development

In the future Shoebox will support back-end cloud services like [AWS Pub/Sub Messaging](#) and [Dynamo DB](#), which would enable unlimited scalability. New storage backends can be added easily to Shoebox by implementing the [Store](#) interface.

7.4 Other Databases

Kweb doesn't require you to use Shoebox. You're free to use any database, either directly, or via a database abstraction layer such as [Exposed](#).

7.5 Working Example

For a more complete example of using Shoebox for persistent storage see the [to do demo](#).

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

8.1 Getting started

First tell Kweb to use the Fomantic UI plugin:

```
import kweb.plugins.fomanticUI.*

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

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

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>
```

This translates to the Kotlin:

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

fun main() {
    Kweb(port = 16097, plugins = listOf(fomanticUIPlugin)4) {
        div(fomantic.ui.icon.input).new {
```

(continues on next page)

(continued from previous page)

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

Take a look at the [Fomantic UI documentation](#) to see everything else it can do.

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/>

Frequently Asked Questions

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

No, Kweb's `immediate events` allows you to avoid any server communication delay by responding immediately to DOM-modifying events.

Kweb is designed to be efficient by default, minimizing both browser and server CPU/memory.

If you encounter a situation in which Kweb is slow please [submit a bug](#).

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, but there are also important differences:

- Kweb is *far* more lightweight than Vaadin. At the time of writing, `kweb-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.

You can see a number of other example Kweb projects here: kwebio/kweb-demos

9.4 How do I enable HTTPS?

Very easily, please see [this example](#).

9.5 Can I embed Kweb within an Android app?

Yes! Please see kweb-demos/tree/master/android for an example.

9.6 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.

9.7 Why risk my project on a framework I just heard of?

Picking a framework is stressful. Pick the wrong one and perhaps the company behind it goes out of business, meaning your entire app is now built on something obsolete. We've been there.

Kweb's development is driven by a community of volunteers. We welcome contributions from anyone, but Kweb doesn't depend on any sponsoring company.

Because of the powerful abstractions it's built on, Kweb also has the advantage of simplicity (<5k loc). This makes it easier for people to contribute, and less code means fewer bugs.

That said, Kweb is still pre-1.0, one of the implications being that we can and will make breaking API changes, and new releases are quite frequent.

9.8 How is "Kweb" pronounced?

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

9.9 Can Kweb be embedded in an existing Ktor app?

Yes! Please see [this example](#).

9.10 I have a question not answered here

Feel free to [ask us a question](#) on Github Issues.