February 2015

Widok User Manual

Tim Nieradzik
University of Bremen
tim@kognit.io

Abstract Widok is a reactive web framework for Scala.js. It enables you to develop interactive web applications entirely in Scala by transpiling the code to JavaScript. Abstractions such as *views*, *channels* or *aggregates* allow for concise and reliable code. Widok ships native bindings for popular CSS frameworks like Bootstrap and Font-Awesome which let you iterate faster.

Contents

Chapter 1

Introduction

Widok is a reactive web framework for Scala.js. Its key concepts are:

- **Page**: the browser query is dispatched to a page which renders widgets
- **Widget**: an element that corresponds to a node in the DOM
- **Channel**: a stream of values
- **Aggregate**: a channel container

1.1 Comparison

Widok is different from traditional web frameworks in the following aspects:

- The rendering logic is implemented entirely on the client-side
- The sole purpose of the server is to exchange data with the client
- Instead of writing HTML templates, widgets are defined in pure Scala code
- Only widgets are allowed to manipulate the DOM
- Designed with memory-efficiency in mind
- Bootstrap 3 bindings available

As Widok is built around Scala.js it also inherits some of its properties:

- IDE support
- Browser source maps
- Fast compilation times

1.2 Data propagation

Channels model continuous values as streams. These streams can be observed. Internally, no copies of the produced values are created. If desired, the current value can be explicitly cached, though. It is possible to operate on channels with higher-order functions such as `map()` and `filter()`. Every time a new value is produced, it is propagated down the observer chain.

Aggregates are channel containers. They allow to deal with large lists efficiently. If an item gets added, removed or updated, this is reflected directly by a change in the DOM, only operating on the actual nodes.

Chapter 2

Getting Started

2.1 First project

We recommend the use of SBT for compilation as it supports continuous compilation and is the official build system used by Scala.js. If you want to use an IDE, Widok is well-supported by IntelliJ.

Create a new directory `project` with two files:

- `plugins.sbt`

```
logLevel := Level.Warn

addSbtPlugin("org.scala-lang.modules.scalajs" % "scalajs-sbt-plugin"
```

- `Build.scala`

```
import sbt._
import sbt.Keys._
import scala.scalajs.sbtplugin.ScalaJSPlugin._

object Build extends sbt.Build {
  val projectName = "example"
  val buildOrganisation = "org.widok"
  val buildVersion = "0.1-SNAPSHOT"
  val buildScalaVersion = "2.11.2"
  val buildScalaOptions = Seq(
    "-unchecked", "-deprecation",
    "-encoding", "utf8",
    "-Xelide-below", annotation.elidable.ALL.toString)

  lazy val main = Project(id = projectName, base = file("."))
    .settings(scalaJSSettings: _*)
    .settings(
      libraryDependencies ++= Seq(
        "io.github.widok" %%% "widok" % "0.1.3"
      ),
```

```
      organization := buildOrganisation,
      version := buildVersion,
      scalaVersion := buildScalaVersion,
      scalacOptions := buildScalaOptions,
      ScalaJSKeys.persistLauncher := true
    )
}
```

The source code goes underneath `src/main/scala/org/example`.

Create a source file `Main.scala` with the following contents:

```
package org.widok.example

import org.widok._
import org.widok.bindings.HTML

object Main extends PageApplication {
  def contents() = Seq(
    HTML.Heading.Level1("Welcome to Widok!"),
    HTML.Paragraph("This is your first application."))

  def ready() {
    log("Page loaded.")
  }
}
```

Finally, you need to create an HTML file `application.html` in the root directory which includes the compiled JavaScript sources:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Widok example</title>
  </head>
  <body id="page">
    <script type="text/javascript" src="./target/scala-2.11/example-fastopt.js"></sc
    <script type="text/javascript" src="./target/scala-2.11/example-launcher.js"></s
  </body>
</html>
```

To compile your application application, run:

```
$ sbt fastOptJS
```

Now you can open `application.html` in your browser.

2.2 Troubleshooting

If sbt does not find the shapeless dependency, try to add the following resolver:

```
resolvers += "bintray-alexander_myltsev" at "http://dl.bintray.com/al
```

See also https://github.com/widok/todomvc/issues/1

2.3 Compilation

The latest version is always published to Sonatype and Maven Central. Therefore, no manual compilation of Widok is required. Please refer to Developing if you would like to try out the latest development release.

Chapter 3

Concepts

In this chapter we will mention all key concepts of Widok. The following chapters will deal with these topics in detail.

3.1 Continuous compilation

SBT can detect changes in source files. It recompiles the project only when needed:

```
$ sbt ~fastOptJS
```

3.2 Single-page applications

The basic application from the previous chapter looked like this:

```
package org.widok.example

import org.widok._
import org.widok.bindings.HTML

object Main extends PageApplication {
  def contents() = Seq(
    HTML.Heading.Level1("Welcome to Widok!"),
    HTML.Paragraph("This is your first application."))

  def ready() {
    log("Page loaded.")
  }
}
```

This is a single-page application. The two methods `contents()` and `ready()` must be implemented. `contents()` is a list of widgets that are rendered when the page is loaded. Afterwards, `ready()` gets called.

3.3 Multi-page applications

While for small applications a single-page approach may be sufficient, you should consider using a router and split the application into multiple pages.

```
package org.widok.example

import org.widok._

object Routes {
  val main = Route("/", pages.Main())
  val test = Route("/test/:param", pages.Test())
  val notFound = Route("/404", pages.NotFound())

  val routes = Set(main, test, notFound)
}

object Main extends RoutingApplication(Routes.routes, Routes.notFound)
```

The `Routes` objects defines all available routes. A query part may also be a named parameter. If the query parameters need to be validated, this should be done in the page itself.

Create a new file for each page:

- `pages/Main.scala`

```
package org.widok.example.pages

import org.widok._
import org.widok.Widget
import org.widok.bindings.HTML
import org.widok.example.Routes

case class Main() extends Page {
  def contents() = HTML.Anchor("Link to second page").url(Routes.test("param", "firs

  def ready(route: InstantiatedRoute) {
    println("Page 'main' loaded.")
  }
}
```

Contrary to single-page applications, `ready()` expects a parameter which contains information about the chosen route and its parameters.

A route can be instantiated with parameters (`Routes.ROUTE(params)`). This method is overloaded. If a route has more than one parameter, a map must be passed instead.

`HTML.Anchor()` is a widget that creates a link. In the above example the target path is set to an instantiated route. This is to be preferred over creating links with hand-written paths. Using instantiated routes ensures during compile-time that no invalid routes are referenced. During runtime, assertions will even check whether the correct parameters were specified.

- pages/Test.scala

```
package org.widok.example.pages

import org.widok._

case class Test() extends Page {
  val query = Channel[String]()

  def contents() = Seq("Received parameter: ", query)

  def ready(route: InstantiatedRoute) {
    query := route.args("param")
  }
}
```

We are registering a channel and pass it the current query parameter. A channel can be considered as a stream you can send data to. The data then gets directly multiplexed to the subscribers. It is used as a widget in `contents()`. This subscribes to the channel and whenever a new value is sent to it, it gets rendered automatically without any caching involved.

Each page is instantiated only once (in `Routes`). If the user changes the URI parameter, the router detects this and calls `ready()` again. This feeds the channel the new parameter. The actual HTML page never gets reloaded.

- pages/NotFound.scala

```
package org.widok.example.pages

import org.scalajs.dom
import org.widok._
import org.widok.bindings.HTML
import org.widok.example.Routes

case class NotFound() extends Page {
  def contents() = HTML.Heading.Level1("Page not found")

  def ready(route: InstantiatedRoute) {
    dom.setTimeout(() => Routes.main().go(), 2000)
  }
}
```

In the router this page was set as a fall-back route and is loaded if no other page matches (or is loaded explictly). Here we are showing how to call JavaScript functions using the DOM bindings. It calls `go()` on the `main` route after two seconds which redirects to it.

3.4 Widgets

The method `contents()` must return a sequence of widgets. When the page is loaded, Widok looks for the element `page` in the DOM and adds all rendered widgets from `contents()` to it.

Widok defines a couple of implicits to make your code more concise. For example, if there is only one element you could drop the sequence and write:

```
def contents() = HTML.Paragraph("text") // <p>text</p>
```

All widgets that expect children actually expect widgets themselves. Therefore, Widok also provides an implicit to convert strings. As in the above example, most channels are converted as well.

The most notable difference to the traditional approach is that instead of writing HTML code you are dealing with type-safe widgets. Widok provides widget bindings for most HTML tags and even custom bindings for Bootstrap. While it is technically possible to embed HTML code (`HTML.Raw()`) and access the elements using `getElementById()` as in JavaScript, this is discouraged as Widok provides better ways to interact with elements.

3.5 Pages

As above, it is advisable to put all pages in a package as to separate them from models and custom widgets.

Pages contain the whole layout. To prevent duplication, custom widgets or partials should be created. This becomes necessary when depending on the user devices different widgets shall be rendered.

For example, Bootstrap splits a page into header, body and footer. You could create a trait `CustomPage` that contains all shared elements like header and footer and requires you only to define `body()` in the pages.

Chapter 4

Build process

Widok uses sbt as its build system.

4.1 Development releases

As with most programming languages, code optimisations are time-consuming and negligible during development. To compile the project without optimisations, run the following sbt command:

```
$ sbt fastOptJS
```

Note that prefixing `~` makes sbt constantly check for changed source files and only recompiles when needed.

This generates two files in `target/scala-2.11/`:

- `APPNAME-fastopt.js`
- `APPNAME-launcher.js`

The former contains the whole project including its dependencies within a single JavaScript file, while the latter is a call to the entry point (see also chapter on applications).

It is safe to concatenate these two files and ship them to the client.

4.2 Production releases

Scala.JS uses the Google Closure Compiler to apply code optimisations. To create an optimised build, run:

```
$ sbt fullOptJS
```

Similarly as with `fastOptJS`, you can also prefix `~` here.

TBD Explain how to have maintain sbt setups for production and development releases. This is necessary in order to have use a different value for `-Xelidable-below`.

4.3 Links

For more information on the build process, please refer to the Scala.JS manual.

Chapter 5

Applications

Although Widok provides certain abstractions for the DOM, it is not necessarily restricted to the browser or to displaying widgets. It was conceived with modularity in mind. Its libraries could also be beneficial in console applications or test cases.

5.1 Entry point

Consider you have a one-file project consisting of:

```
object Main extends Application {
  def main() {
    stub()
  }
}
```

An object of `Application` always defines the entry point of the application. There can only be one entry point.

Apparently, no browser-related functionality was used by the above code. Therefore, it also runs under node.js:

```
$ sbt fastOptJS
$ cat target/scala-2.11/*js | node
stub
```

You can also open it in the browser and as expected it will print `stub` in the browser console.

5.2 Application providers

To reduce the amount of code needed to get started with Widok, it ships two application providers:

- `PageApplication`: A single-page application
- `RoutingApplication`: A multi-page application with routing

5.2.1 Single-page applications

An example for `PageApplication` was already given in the chapter Getting started. The only difference over to a raw `Application` is that you need to implement the two methods `contents()` and `ready()`.

5.2.2 Multi-page applications

For multi-page applications you can use the default router as the application's entry point. In order to do so, create an object of `RoutingApplication` with two arguments: the set of enabled routes and a fall-back route. See the chapter on Router for more information.

Chapter 6

Router

When dealing with applications that consist of more than one page, a routing system becomes inevitable.

The router observes the URL's fragment identifier. For example in `application.html#` the part after the hash symbol `/page` denotes the fragment identifier. A router is initialised with a set of routes which defines all addressable pages. A fallback route may also be specified.

6.1 Interface

The router may be used as follows:

```
object Main extends Application {
    val main = Route("/", pages.Main())
    val test = Route("/test/:param", pages.Test())
    val test2 = Route("/test/:param/:param2", pages.Test())
    val notFound = Route("/404", pages.NotFound())

    val routes = Set(main, test, notFound)

    def main() {
        val router = Router(enabled, fallback = Some(notFound))
        router.listen()
    }
}
```

`routes` denotes the list of enabled routes. It should also contain the `notFound` route. Otherwise, this route could not be loaded using `#/404`.

6.2 Routes

To construct a new route, pass the path and the page object to `Route()`. Page routes may be overloaded with different paths as above with `test` and `test2`.

A part of a path is the contents separated by a slash. For instance, the `test` route above has two parts: `test` and `:param`. A part beginning with a colon

is a placeholder. It extracts the respective value from the URL's fragment and binds it to the placeholder name. Note that a placeholder always refers to the whole part.

A route is said *instantiated* when it gets called:

```
// Zero parameters
val instMain: InstantiatedRoute = Main.main()

// One parameter
val instTest: InstantiatedRoute = Main.test("param", "value")

// Multiple parameters
val instTest2: InstantiatedRoute = Main.test2(Map(
    "param" -> "value",
    "param2" -> "value2"))

// Change the current page to /test/value
instTest.go()
```

To query the instantiated parameters, access the `args` field in the first parameter passed to `ready()`.

```
case class Test() extends Page {
  ...
  def ready(route: InstantiatedRoute) {
    log(route.args("param"))

    // Accessing optional parameters with get()
    // This returns an Option[String]
    log(route.args.get("param2"))
  }
}
```

6.2.1 Motivation

Due to the simple design the router could be efficiently implemented. The routes allow better reasoning than it would be possible if they supported regular expressions. When the router is constructed, it sorts all routes by their length and checks whether there are no conflicts. Also, the restriction that each parameter must be named makes code more readable when referring to parameters of an instantiated route. If validation of parameters is desired, this must be done in `ready()`. The advantages of the simple design outweigh its limitations.

6.3 Application provider

As the router defines usually the entry point of an application, Widok provides an application provider that enforces better separation:

```
object Routes {
  val main = Route("/", pages.Main())
  ...
  val notFound = Route("/404", pages.NotFound())

  val routes = Set(main, ..., notFound)
}

object Main extends RoutingApplication(Routes.routes, Routes.notFound
```

This is to be preferred when no further logic should be executed in the entry point prior to setting up the router.

Chapter 7

Widgets

A widget represents an element to be displayed by the browser. Instances of widgets can be nested while enforcing type-safety. Custom widgets can be defined. The idea here is to compose a widget of smaller, existing ones.

7.1 HTML

Widok provides widgets for many HTML elements. The bindings have a more intuitive naming than their HTML counterparts. The module they reside in is `org.widok.bindings.HTML`. Although it is possible to import the whole contents, it is advisable to address the widgets using a qualified access. Usually, a project needs to define its own widgets which most likely will shadow widgets from the HTML bindings. Instead, the custom widgets could be imported into the namespace. These custom widgets in turn will depend on the HTML widgets and only extend them with CSS tags for instance.

Tag	Widget	Notes
h1	Heading.Level1	
h2	Heading.Level2	
h3	Heading.Level3	
h4	Heading.Level4	
h5	Heading.Level5	
h6	Heading.Level6	
p	Paragraph	
b	Text.Bold	
small	Text.Small	
span	Raw	sets `innerHTML`
img	Image	
br	LineBreak	
button	Button	
section	Section	
header	Header	
footer	Footer	
nav	Navigation	

Tag	Widget	Notes
a	Anchor	
form	Form	
label	Label	
input	Input.Text	sets `type="text"`
input	Input.Checkbox	sets `type="checkbox"`
select	Input.Select	
ul	List.Unordered	
ol	List.Ordered	
li	List.Item	
div	Container.Generic	
span	Container.Inline	

7.1.1 Usage

A widget is always an instance of `Widget` and can be used like a function:

```
val widget = HTML.Raw("<b><i>Text</i></b>")

// This is equivalent to:
val widget2 = Text.Bold(
    Text.Italic("Text"))
```

Most widgets either expect parameters or children. However, there are also widgets which expect both:

```
Anchor("http://en.wikipedia.org/")(
    Text.Bold("Wikipedia"))
```

Hint: Use the code completion of your IDE to figure out which widgets are available and which parameters to pass.

7.2 Bootstrap

Here is an example on how to use the Bootstrap bindings:

```
package org.widok.example.pages

import org.widok._
import org.widok.bindings.HTML
import org.widok.example.Routes
import org.widok.bindings.Bootstrap._

case class Main() extends Page {
  def header() =
    NavigationBar()(
```

```
      Container(
        NavigationBar.Header(
          NavigationBar.Toggle(),
          NavigationBar.Brand("Application name")),
        NavigationBar.Collapse(
          NavigationBar.Elements(
            NavigationBar.Leaf(Routes.notFound())(
              Glyphicon(Glyphicon.Search), "Page 1"),
            NavigationBar.Leaf(Routes.notFound())(
              Glyphicon(Glyphicon.Bookmark), "Page 2")),
          NavigationBar.Right(
            NavigationBar.Navigation(
              NavigationBar.Form(
                FormGroup(Role.Search)(
                  InputGroup(
                    Input.Text(placeholder = "Search query…")),
                  Button(Glyphicon.Search)()))))))

  def contents() = Seq(
    header(),
    Container(
      PageHeader(HTML.Heading.Level1("Page title ", HTML.Text.Small("
      Lead("Lead text"),
      "Page body"),
    Footer(Container(MutedText("Example page - All rights reserved.")

  def ready(route: InstantiatedRoute) {}
}
```

As probably more than one page is going to use the same header, you should create a trait for it. You may also want to write a trait `CustomPage` which only requires you to define the page title and body in every page.

For the bindings to work, add the latest Bootstrap stylesheet to the `head` tag of your `application.html` file. If you want to use a CDN, use:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstra
```

7.3 Creating custom widgets

Widgets should be designed to be restrictive. For example, the only children `List.Unordered()` accepts are instances of `List.Item`. For custom widgets, create a class hierarchy which closely resembles the intended nesting of the elements. This will turn out to be helpful because you implicitly establish type-safety for CSS components. When widgets are changed, this will catch usage errors during compile-time.

A custom widget may be defined as follows:

```
def Panel(contents: Widget*) =
    HTML.Container.Generic(
      HTML.Container.Generic(
        contents: _*
      ).css("panel-body")
    ).css("panel", "panel-default")
```

This corresponds to:

```
<div class="panel panel-default">
    <div class="panel-body">
        ... contents of the widget's children ...
    </div>
</div>
```

7.4 Binding to events

Each widget provides useful functions to better interact with the DOM. Instead of setting IDs on elements and requesting elements using `getElementById()`, we are constantly working with objects which is less error-prone.

To bind to the click event to a button, write:

```
import org.scalajs.dom.MouseEvent

...

Button(Glyphicon.Book)("Show timestamp")
    .bindMouse(Event.Mouse.Click, (e: MouseEvent) => println(e.timeStamp))
```

Note that methods on a widget return the instance of the widget. This allows to arbitrarily nest widgets and change their attributes, without storing a reference to the widget in a local variable.

The above example does not seem all too different from attaching a callback like in JavaScript. This is also what happens implicitly. In fact, the second argument of `bindMouse()` expects a *Channel*, a concept we will introduce in the next chapter. A channel produces data which is passed on to its subscribers. In the above example, a Scala implicit turns the lambda function into a channel. But similarly, we could have written:

```
val click = Channel[MouseEvent]()
click.attach(e => println(e.timeStamp))
...

Button(Glyphicon.Book)("Show timestamp")
    .bindMouse(Event.Mouse.Click, click)
```

The advantage may not be obvious on first sight, but a channel can have multiple subscribers. This is important in web applications where data gets propagated to various layers of the application. For example, consider a shopping cart. Items get modified in the product listing. At the same time the header needs to get updated with the newly calculated price.

Now that `click` is a stream of events, we could decide to take into account only the first event:

```
click.head.attach(e => println(e.timeStamp))
```

Another prominent use case of channels are dynamic changes of widgets, such as the visibility:

```
HTML.Container.Generic("Button clicked")
    .show(click.head.map(_ => false))
```

`show()` expects a boolean channel. Depending on the values that are sent to the channel a widget is shown or not. Here, we hide the widget as soon as we click the button.

The chapter Data propagation deals with channels in detail.

7.5 Links

- API documentation

Chapter 8

Data Propagation

Due to its nature, user interfaces (UIs) are notoriously heavily data-driven. Values do not only need to be displayed once, but continuously modified as the user interacts with the interface.

A higher interactivity results in more data dependencies and ultimately in more complex code. Imperative code in particular is prone to this shortcoming as dependencies must be modelled manually. With web application getting increasingly more interactive, solutions with clean and concise code are needed.

Widok advocates a flow-driven approach: Values are expressed in a stream-like data structure which we call *channel*. A channel is equipped with operations that allow to easily express data dependencies. Channels can have children to which the incoming values are propagated.

Built on top of channels, there is another data structure called *aggregates*. Channels model single values while aggregates represent a collection of values.

A key concept of channels and aggregates is that data may be rendered directly to the DOM without any intermediate caching. Whenever a new value is pushed to a channel, an atomic update takes place, only changing the associated DOM node. Similarly, this is respected when inserting, updating or removing items in an aggregate.

For an application that makes heavy use of data propagation, see our TodoMVC implementation.

The proper functioning of each operation is backed by test cases. These serve as a complementary documentation as only a handful of operations are explained in full detail here.

8.1 Channels

A channel is a multiplexer for typed messages (immutable values) that it receives. Values sent to the channel get propagated to the observers that have been attached to the channel, in the same order as they were added.

As outlined above, channels are a memory-efficient model and may be bound to widgets. The widget renders the received values on-the-fly. However, the lack of state restricts the possibilities. It may be desired to obtain the current value of a channel, change it or perform more elaborate operations that inherently

require caching. This was taken into consideration and explicit caching can be performed.

Here is a simple example for a channel that receives integers and prints these on the console:

```
val ch = Channel[Int]()  // initialise
ch.attach(println)       // attach observer
ch := 42                 // produce value
```

Note: The `:=` operator is a shortcut for the method `produce`.

Channels define useful operations to express data dependencies:

```
val ch = Channel[Int]()
ch.filter(_ > 3)
  .map(_ + 1)
  .attach(println)
ch := 42 // 43 printed
ch := 1  // nothing printed
```

These operations are well-known methods such as `map()`, `filter()`, `take()` etc. As they return instances of `Channel`, they can be chained as in the example.

A particular operator defined on channels is the addition. By adding up channels a new container channel is constructed. All values produced by the operands are propagated to this resulting channel and vice-versa (two-way propagation), but not amongst the operands:

```
val cont = ch + ch2  // more operands are possible, too
ch  := 42            // propagated to cont
ch2 := 23            // propagated to cont
cont := 65           // propagated to ch and ch2
```

8.1.1 State channels

State channels are channels that have an initial value. In reactive programming, these are known as *cold observers*.

The value that the state channel was instantiated with gets propagated upon attaching to the channel. A common use case are user interfaces. Usually, channels are set up before the widgets. The channels are then bound to the widgets. This renders the initial value directly in the DOM. Otherwise, it would be necessary to send the initial value to the channel manually as soon as the widget was rendered.

The following example visualises the difference in behaviour:

```
val ch = Channel.unit(42)
ch.attach(println) // prints 42

val ch2 = Channel[Int]()
```

```
ch2 := 42 // lost as ch2 does not have any observers
ch2.attach(println)
```

> **Note**: `Channel.unit(value)` is equal to `StateChannel(value)`.

The argument is evaluated lazily and can also point to a mutable variable:

```
var counter = 0
val ch = Channel.unit(counter)
ch.attach(value => {counter += 1; println(value)}) // prints 0
ch.attach(value => {counter += 1; println(value)}) // prints 1
```

The following example illustrates a conceptual issue that arises with the use of chaining:

```
val ch = Channel.unit(42)
val ch2 = ch.map(_ + 1)
ch2.attach(...) // produces 43?
ch2.attach(...) // produces 43?
```

The question is whether initially produced values get propagated along the chain of operations.

8.1.2 Child channels

In order to make all operations work properly on state channels, the notion of *child channels* was introduced. The previously given example is therefore well-defined. A child channel delays the propagation of the initial value until an observer is attached.

In order to figure out whether an operation supports this behaviour, it is sufficient to investigate its return type. The child channel behaviour only makes sense on operations that do not need more than one produced value to propagate it. A counterexample is `skip()` which conceptually must skip at least one element; the initial value will therefore never be propagated:

```
val ch = Channel.unit(42)
ch.skip(1) // returns Channel
  .attach(...)
```

> **Note**: Higher-order functions must not have any side-effects when used in operations on state channels.
>
> ```
> val ch = Channel.unit(42)
> ch.map(println).attach(_ => ())
> ```
>
> The example does not behave as expected and prints 42 two times instead of only once. This design decision was made consciously as to keep the implementation of operations more simple.

8.1.3 Cached channels

For better performance, channels do not cache the produced values. Some operations cannot be implemented without access to the current value, though. Therefore, *cached channels* were introduced.

For example, `update()` is an operation that would not work without caching. It takes a function which modifies the current value:

```
val ch = CachedChannel[Test]()
ch.attach(println)
ch := 2
ch.update(_ + 1) // produces 3
```

`value()` is another helpful operation. It creates a channel lens. If a channel is made up of a `case class`, you could obviously use `.map(_.field)` to obtain a channel for a field. However, as with lenses in functional programming, a back channel is desired which composes a new value with this field changed. This also works with nested values as in the following example:

```
case class Base(a: Int)
case class Test(a: String, b: Base)

val ch = CachedChannel[Test]()
val lens = ch.value[Int](_ >> 'b >> 'a)

ch.attach(println)

ch := Test("hello world", Base(1))
lens := 2 // produces Test("hello world", Base(2)) on ch
```

In order to get a cached version of an existing channel, use the method `cache`:

```
val cache = ch.cache
ch := 1
cache.update(_ + 1) // produces 2 on ``ch``
```

Use the method `unique` to produce a value if it is the first or different from the previous one. A use case is sending HTTP requests upon user input:

```
val ch = cache.unique
ch.attach { query =>
    // perform HTTP request
}
```

> **Note**: As a `CachedChannel` inherits from `Channel`, all channel operations are available as well.

8.1.4 Widgets

A channel can be connected with one or more widgets. Most widgets provide two-way binding. To bind a channel to a widget, use the method `bind()` or its specialisations `bindRaw()`, `bindWidget()` etc.

When associating a channel to multiple widgets, the contents amongst them is synchronised automatically:

```
val ch = Channel.unit("Hello world")
def contents() = Seq(
  Input.Text().bind(ch, live = true),
  Input.Text().bind(ch)
)
```

This creates two text fields. When the page is loaded, both have the same content: "Hello world". If the user changes the text of the first field, the second text field is updated on-the-fly. The second field has the live mode disabled and an enter press is required before the change gets propagated to the first text box.

> **Note:** `bind()` may only be used once. If you want to have multiple event handlers, you can chain these using the `+` operator that was mentioned above.

Channels are consistently used for all kind of DOM events. This allows to connect even two unrelated widgets. The following example connects a text field with a button:

```
val query = Channel[String]()
val click = Channel[Unit]()

query.attach(println)

def contents() = Seq(
  Input.Text().bind(query, click),
  Button().bind(click)
)
```

The second parameter of `bind()` on the text field is a channel that "flushes" its contents. Sending a message to it is equivalent to pressing enter in the field. The button is bound to the same channel. Pressing the button results in the text field to produce its contents to `query`. If widgets were using callbacks instead, the above would be a lot harder to implement.

The method `cssCh()` on widgets sets a CSS flag only if the expected boolean channel produces `true` values, otherwise it is unset:

```
widget
 .cssCh(editing, "editing")
```

An implicit is defined so that string channels can be used easily:

```
val name = Channel[String]()
def contents() = Heading.Level1("Hello ", name)
```

`name` is converted into a widget and updates the DOM when new values are produced.

Another implicit is provided for `Channel[Widget]`. You can use `map()` on a channel to create a widget stream. These are rendered automatically. If the widget type is always the same and it provides a `bind()` method, this is to be preferred instead. `bind()` does not recreate the widget itself and is therefore more efficient.

8.2 Aggregates

Aggregates allow to efficiently deal with lists of changing values. An aggregate is implemented as a list of channels.

You may be wondering what is the purpose of having `Aggregate[T]` while `Channel[Seq[T]]` could be used. If you need to operate on a subsets, it is hard to do so with the latter approach. What's more, it is also a costly operation. If, however, the elements of your list are constant, you should use the channel approach. Widgets provide `bind()` for both variants.

```
val agg = Aggregate[Int]()
val ch = agg.append() // adds a new row; returns Channel
ch := 42              // sets row value to 42
```

As with channels, an added row is lost if no observer was added beforehand. In practice, attaching a self-written observers is not as common as for channels. Widok already provides important operations and aggregates are most useful in connection with widgets. The general recommendation is to never use `attach()` to keep the code clean. Observers can get messy, especially if back-propagation is involved. If an observer becomes necessary, in almost every case this indicates that an operation should be implemented instead. Refer to the test cases for more information on writing observers for aggregates.

To observe the size of an aggregate, use `size` which returns a `Channel[Int]`:

```
agg.size.attach(println)
```

As aggregates are just containers for channels, it is possible to obtain sub-lists and propagate changes back:

```
val agg = Aggregate[Int]()
val filter = agg.filter(_ % 2 == 0)

agg.append(3)  // not propagated
agg.append(4)  // propagated
agg.append(5)  // not propagated
```

```
filter.clear() // back-propagates the deletion to agg
               // agg will only contain 3 and 5.
```

Note: Not every operation implements back-propagation.

8.2.1 Cached aggregates

Similarly to channels, there is a cached counterpart which stores the most recent value for each row.

```
val agg = Aggregate[Int]()
val cached = agg.cache
```

The method `filterCh()` expects a channel that is producing filter functions:

```
val f = Channel[Int => Boolean]()
val filtered = cached.filterCh(f)

f := _ > 1
f := _ > 2
```

Every time a new filter function is produced to `f`, the filter is applied on all elements from `agg` and `filtered` subsequently only contains those matching ones.

8.2.2 Widgets

All list-like widgets (such as tables) provide the method `bind()`:

```
List.Unordered().bind(agg) { ch =>
  List.Item(ch)
}
```

Aggregates implement many operations which interact nicely with widgets: `isEmpty` could be used to hide widgets if an aggregate is empty:

```
val agg = Aggregate[Int]()

val widget = HTML.Container.Inline("The list is empty.")
  .show(agg.isEmpty)

val widget2 = HTML.Container.Inline("The list is not empty.")
  .show(agg.nonEmpty)
```

8.3 Debugging

Although, channels and aggregates are used by widgets, they do not depend on JavaScript features. You can therefore debug their use directly on the console:

```
$ sbt console
import org.widok._
val ch = Channel[Int]()
ch.attach(println)
ch := 42
```

8.4 API documentation

- Channel
- Aggregate

Chapter 9

Developing

9.1 API

Widok is still in its early stages and the API may be subject to changes. Any recommendations for improvements are welcome.

9.2 Compilation

To work on the development version of Widok, run the following commands:

```
$ git clone git@github.com:widok/widok.git
$ cd widok
$ sbt publish-local
```

This compiles the latest version of Widok and installs it locally. To use it, make sure to also update the version in your project accordingly.

9.3 Releases

The manual must always reflect the latest version.