

March 2015

Widok User Manual v0.2

Tim Nieradzik

University of Bremen

`tim@kognit.io`

Abstract Widok is a reactive web framework for the JVM and Scala.js. It enables you to develop interactive client-server applications entirely in Scala. The client code is transpiled to JavaScript, while sharing the same interfaces on the server. Abstractions such as *widgets* and reactive data structures allow for concise and reliable code. Widok ships native bindings for popular CSS frameworks which let you iterate faster.

Contents

Contents	2
1 Introduction	5
1.1 Comparison	5
2 Getting Started	7
2.1 Prerequisites	7
2.2 Project structure	7
2.3 Code	8
2.4 Compilation	9
3 Concepts	11
3.1 Basic application	11
3.2 Single-page applications	11
3.3 Multi-page applications	12
3.4 Pages	14
3.5 Widgets	15
3.6 Reactive programming	15
4 Build process	17
4.1 JDK	17
4.2 Development releases	17
4.3 Production releases	18
4.3.1 Additional optimisations	18
4.4 Continuous compilation	18
4.5 Configure paths	19
4.6 sbt-web	19
4.6.1 Sass	20
4.6.2 Font-Awesome	21
4.6.3 Artifacts	21
4.7 Code sharing	21
4.8 Colours	22
5 Router	23

5.1	Interface	23
5.2	Routes	23
5.2.1	Design decisions	24
5.3	Application provider	25
6	Widgets	27
6.1	HTML	27
6.1.1	Aliases	28
6.2	Usage	28
6.3	Writing custom widgets	29
6.4	Binding to events	30
6.5	Composed widgets	30
6.6	Implicits	31
7	Reactive programming	33
7.1	Motivation	33
7.2	Concepts	33
7.3	Requirements	35
7.4	Implementation	36
7.5	Reactive data structures	39
7.6	Channels	40
7.6.1	Operations	40
7.6.2	State channels	41
7.6.3	Call semantics	43
7.7	Buffers	43
7.8	Dictionaries	45
7.9	Sets	45
7.10	Binding to Widgets	45
7.11	Tests	46
8	Bindings	47
8.1	Bootstrap	47
8.1.1	External stylesheet	47
8.1.2	Label	48
8.1.3	Glyphicons	48
8.1.4	Forms	48
8.1.5	Layout	49
8.1.6	Navigation	50
8.1.7	Navigation bar	50
8.1.8	Alert	51
8.1.9	Progress bar	51
8.1.10	Panel	51
8.1.11	Pagination	51
8.1.12	List groups	51
8.1.13	Grids	52
8.1.14	Modal	52

- 8.1.15 Media 52
- 8.1.16 Breadcrumb 52
- 8.1.17 Table 53
- 8.1.18 Typeahead 53
- 8.2 Font-Awesome 53

- 9 Developing 55**
 - 9.1 API 55
 - 9.2 Compilation 55
 - 9.3 Releases 55
 - 9.4 Manual 55

- 10 Changelog 57**
 - 10.1 Version 0.2 57
 - 10.1.1 General work 57
 - 10.1.2 Event propagation 57
 - 10.1.3 Widgets 58
 - 10.1.4 Routing 59
 - 10.1.5 Trivia 60

- 11 Support 61**

INTRODUCTION

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

- **Pages:** Widok enforces modularisation of your web application. You split your application into *pages*. A *router* watches the browser query and loads the respective page.
- **Widgets:** The layout is specified in terms of composable widgets. Widgets are bound to dynamically changing values which are rendered on-the-fly in the browser.
- **Bindings:** Widok ships bindings for CSS frameworks like Bootstrap and Font-Awesome.
- **Reactive programming:** Reactive data structures, which implement a simple model of data propagation, are provided. Widok has reactive counterparts for variables, arrays, maps and sets. Instead of dealing with constant values, you specify the data flow as streams, on which you operate with higher-order functions like `map()` or `filter()`.

1.1 Comparison

In contrast to traditional web frameworks, a Widok application would implement the entire rendering logic and user interaction on the client-side. The sole purpose of the server would be to exchange data with the client. This approach leads to lower latencies and is more suitable for interactive applications.

Instead of writing HTML templates and doing manual DOM manipulations, Widok advocates widgets which are inspired by traditional GUI development. Widgets are first-class objects, allowing you to return them in functions. This can be useful when rendering a data stream consisting of widgets, or when you want to display a different widget depending on the device the client is using.

Another strength of Widok is that you can develop client-server applications entirely in Scala and CSS. Scala.js transpiles your code to JavaScript. Having only one implementation language reduces redundancy due to code sharing. This is especially useful for protocols. It also lets you develop safer web applications since you could use the same validation code on the client as on the server.

Widok is fully supported by IntelliJ IDEA. As Scala is a statically typed language you can use your IDE for refactoring and tab completion, increasing your productivity. Similarly, many bugs can be already identified during compile-time. Browser source maps will help you pinpoint run-time errors in the original source code. Scala.js supports continuous compilation which lets you iterate faster.

Finally, Widok is not necessarily bound to web applications. As it compiles to regular JavaScript code, you could develop [io.js](#) applications or even native user interfaces with [NW.js](#). The JVM build comprises the reactive library, so that you can use it on the server-side as well.

GETTING STARTED

This chapter will guide you through creating your first Widok project.

2.1 Prerequisites

To develop web applications with Widok the only dependency you will need is `sbt`. Once installed, it will automatically fetch `Scala.js` and all libraries Widok depends on.

You may also want to use an IDE for development. Widok is well-supported by `IntelliJ IDEA` with the `Scala plugin`. The use of an IDE is recommended as the interfaces Widok provides are fully typed, which lets you do tab completion.

2.2 Project structure

Your project will have the following structure:

```
application.html
project
  Build.scala
  plugins.sbt
src
  main
    scala
      example
        Application.scala
```

Create a directory for your project. Within your project folder, create a sub-directory `project` with the two files `plugins.sbt` and `Build.scala`:

- `plugins.sbt` specifies sbt plug-ins, notably `Scala.js`

```
logLevel := Level.Warn
```

```
addSbtPlugin("org.scala-js" % "sbt-scalajs" % "0.6.0")
```

- `Build.scala` is the build configuration of your project. The configuration itself is specified in Scala code, which allows for more flexibility. The chapter ‘Build process’ explains some of the possibilities in the web context.

```
import sbt._
import sbt.Keys._
import org.scalajs.sbtplugin._
import org.scalajs.sbtplugin.ScalaJSPlugin.autoImport._

object Build extends sbt.Build {
  val buildOrganisation = "example"
  val buildVersion = "0.1-SNAPSHOT"
  val buildScalaVersion = "2.11.6"
  val buildScalaOptions = Seq(
    "-unchecked", "-deprecation"
  , "-encoding", "utf8"
  , "-Xelide-below", annotation.elidable.ALL.toString
  )

  lazy val main = Project(id = "example", base = file("."))
    .enablePlugins(ScalaJSPlugin)
    .settings(
      libraryDependencies ++= Seq(
        "io.github.widok" %% "widok" % "0.2.0"
      )
    , organization := buildOrganisation
    , version := buildVersion
    , scalaVersion := buildScalaVersion
    , scalacOptions := buildScalaOptions
    , persistLauncher := true
    )
}
```

Your source code goes underneath `src/main/scala/example/`.

2.3 Code

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

```
package example

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

object Main extends PageApplication {
```



```

def view() = Inline(
  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. It references 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"
    ></script>

    <script
      type="text/javascript"
      src="./target/scala-2.11/example-launcher.js"
    ></script>
  </body>
</html>

```

2.4 Compilation

This is all you need for a minimal Widok project. To compile your application, run:

```
$ sbt fastOptJS
```

Now you can open `application.html` in your browser. The page should show a heading with a paragraph. Obviously, the Scala code you wrote translates to:

```

<h1>Welcome to Widok!</h1>
<p>This is your first application.</p>

```

Upon page load this gets dynamically inserted into the node with the ID `page`. When you open up the browser's web console, it will show the message you specified in `ready()`.

CONCEPTS

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

3.1 Basic application

Consider a one-file project consisting of:

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

A global object of type `Application` defines the entry point of the application¹. You could use methods from Widok's DOM object to access and modify the DOM.

Compile this application:

```
$ sbt fastOptJS
```

Open your `application.html` in the browser and it will print `stub` in the web console. The example doesn't use any browser-related functionality. Therefore, it would also run under `io.js`.

```
$ cat target/scala-2.11/*js | iojs  
stub
```

3.2 Single-page applications

The application from the previous chapter roughly looked like this:

```
package example  
  
import org.widok._
```

¹An application cannot define more than one entry point.

```
object Main extends PageApplication {
  def view() = Inline()
  def ready() { }
}
```

For a single-page application you need to declare an object which inherits from `PageApplication`, whereby `Scala.js` knows that it shall be the entry-point of the program.

Furthermore, the two methods `view()` and `ready()` must be implemented. The views are rendered when the page is loaded. Afterwards, `ready()` gets called.

Note: The `Inline()` view is a container that groups multiple widgets without affecting the design².

3.3 Multi-page applications

While for small applications a single-page approach may be sufficient, you should consider making use of the in-built router and split your application into multiple pages for better modularity:

```
package 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
)
```

A multi-page application must extend `RoutingApplication` which is passed a list of routes and a fallback route. Here, the `Routes` objects defines the available routes. The query part of a route can be parameterised by prefixing a colon. For instance, `param` is a named parameter for the route `test`. The router only matches strings. Further validations and conversions could be performed in the page itself.

²In contrast to a `` or `<div>`, an `Inline` view cannot be controlled using CSS stylesheet rules.

Create a new file for each page:

- pages/Main.scala

```
package example.pages

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

import example._

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

  def ready(route: InstantiatedRoute) {
    log(s"Page 'main' loaded with route '$route'")
  }

  override def destroy() {
    log("Page 'main' left")
  }
}
```

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

This page uses `HTML.Anchor()` which is a widget representing a link. The target URL is set to an instantiated route, namely `test`. Every route can be instantiated, although all parameters according to the route specification must be provided. The `apply()` method of a route is overloaded. For only one route parameter, the first argument denotes the named parameter and the second its value. If a route has more than one parameter, a map with the values must be passed instead. Instantiating routes is to be preferred over links with hand-written paths. Referencing routes ensures during compile-time that no invalid routes are referred to. During runtime, assertions will verify whether all correct parameters were specified.

When clicking the link, the router will notice this change and render the new route. The actual HTML file of the page is not reloaded, though.

By default, `destroy()` is a stub, but may be overridden when navigating between routes requires resource management.

- pages/Test.scala

```
package example.pages

import org.widok._
```

```

case class Test() extends Page {
  val query = Channel[String]()
  def view() = Inline("Received parameter: ", query)

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

```

Here, we are registering a channel and pass it the current value of the query parameter `param`. A channel can be considered as a stream you can send messages to. Each message is then multiplexed to the subscribers. `query` has one subscriber here. As it is used in `view()`, it is converted into a view. Whenever a new value is produced on `query` (using `:=`), it gets rendered automatically in the browser. If the user changes the query parameter of the current page, the router will detect this and re-render the page.

- `pages/NotFound.scala`

```

package example.pages

import org.scalajs.dom

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

import example.Routes

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

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

```

`NotFound` was set as a fall-back route. It is loaded when no other route matches or when the fall-back route is loaded explicitly. Here, we are showing how to call JavaScript functions using the [DOM bindings](#). It redirects to the main page after two seconds by calling `go()` on the instantiated route.

3.4 Pages

As in the multi-page application, it is advisable to put all pages in a package as to separate them from other parts of the application, like models, validators or partials³.

³Partials are composed widgets.

When the page is loaded, Widok looks for the element with the ID `page` in the DOM and renders `view()` in this node. The entire contents is destroyed when the route changes.

`view()` must return the whole layout of the page. To prevent duplication among pages, partials should be defined. Common candidates for partials are navigation bars, panels or layout elements. But as partials are just regular functions returning a widget, they can contain logic and you may render different widgets depending on whether the user accesses the website on a mobile device or on a desktop.

For instance, Bootstrap splits pages 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.

3.5 Widgets

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 HTML tags, and custom bindings for Bootstrap and Font-Awesome. It is possible to embed HTML code using the `HTML.Raw()` widget. You could even access DOM elements using `DOM.getElementById()` as in JavaScript. However, this is discouraged in Widok which provides better ways to interact with elements.

3.6 Reactive programming

To showcase some of the capabilities of reactive programming for UI development, take the following example:

```
package example

import org.widok._
import org.widok.html._

object App extends PageApplication {
  val name = Var("")
  val hasName = name.map(_.nonEmpty)

  def view() = div(
    h1("Welcome to Widok!")
    , p("Please enter your name:")

    , text().bind(name)

    , p("Hello, ", name)
      .show(hasName)
```

```

, button("Change my name")
  .onClick(_ => name := "tux")
  .show(name.unequal("tux"))

, button("Log out")
  .onClick(_ => name := "")
  .show(hasName)
)

def ready() { }
}

```

The first striking change from the previous examples is that we now use the HTML aliases (`import org.widok.html._`).

More importantly, this example shows that widgets provide methods to interact with channels. For example, the method `bind()` on textual input fields realises two-way binding, i.e., every key stroke produces a new value on the channel and notifies all other subscribers.

Another related method is `show()` which will only show a certain widget if the passed channel produces the value `true`.

`Var()` is a channel with an empty value as a default value and is bound to `name`. Well-known combinators such as `map()` and `filter()` are also defined on channels. In the example, `map()` is used for `hasName` such that the channel notifies its subscribers whenever `name` is updated.

BUILD PROCESS

The chapter ‘Getting Started’ proposed a simple sbt configuration. sbt is a flexible build tool and can be extended with plug-ins and custom tasks. Some useful advice on using sbt for web development is given here.

For more information on the build process, please refer to the [Scala.js documentation](#).

4.1 JDK

The [Oracle JDK](#) leads to slightly shorter compilation times than [OpenJDK](#).

With the default configuration, sbt tends to allocate a lot of memory, so that you may run into out-of-memory situations. This can be mitigated by limiting the heap size with the `JAVA_OPTS` environment variable:

```
export JAVA_OPTS="$JAVA_OPTS \  
-XX:InitialHeapSize=128m \  
-XX:MaxHeapSize=512m \  
-XX:+CMSClassUnloadingEnabled"  
export SBT_OPTS="$JAVA_OPTS"
```

4.2 Development releases

Code optimisations are time-consuming and usually negligible during development. To compile the project without optimisations, use the `fastOptJS` task:

```
$ sbt fastOptJS
```

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

- `$ProjectName-fastopt.js`
- `$ProjectName-launcher.js`

The former is the whole project including dependencies within a single JavaScript file, while the latter contains a call to the entry point. It is safe to concatenate these two files and ship them to the client.

4.3 Production releases

Scala.js uses Google's [Closure Compiler](#) to apply code optimisations. To create an optimised build, use the `fullOptJS` task:

```
$ sbt fullOptJS
```

You may want to add a constant to your sbt configuration to toggle compiler settings depending on whether you need a production or development release. For example, `-Xelidable=below` could be used to remove assertions from production releases for better performance.

4.3.1 Additional optimisations

The Scala.js compiler provides settings to fine-tune the build process.

To further reduce the build size, class names could be replaced by an empty string. The semantics of a program should never rely on class names. This optimisation is therefore safe to set. However, if you want to retain some class names, you could define exceptions, for example for classes from a certain namespace.

Another option is to enable unchecked `asInstanceOf` casts. A cast should always be well-defined. If this cannot be ensured, a manual `assertInstanceOf` check needs to be performed anyway. Expecting an exception to be thrown is a suboptimal way of dealing with potentially undefined casts. Under this assumption, `asInstanceOf` casts should work if unchecked. Scala.js lets you change the semantics for the sake of better performance.

```
import org.scalajs.core.tools.sem._
...
scalaJSSemantics ~= (
  .withRuntimeClassName(_ => "")
  .withAsInstanceOfs(CheckedBehavior.Unchecked)
)
```

Since class names can be useful for debugging purposes and illegal casts may happen during development, these two options should only be set for production releases.

4.4 Continuous compilation

sbt can detect changes in source files and recompile only when needed. To do so, prefix `~` to your build task (either `fastOptJS` or `fullOptJS`), for example:

```
$ sbt ~fastOptJS
```

This leads to faster development cycles than executing `fastOptJS` on your own.

4.5 Configure paths

If the web server should point directly to the most recently built version, you do not need to copy over the generated files each time. Instead, the paths can be customised. A recommended application hierarchy is the following:

- `web/index.html`: Self-written entry-point of the application
- `web/js/`: Generated JavaScript files
- `web/css/`: Generated CSS stylesheets
- `web/fonts/`: A copy of all font files (for example, Bootstrap glyphicons or Font-Awesome)

To do so, specify the paths in the build configuration as follows:

```
val outPath = new File("web")
val jsPath = outPath / "js"
val cssPath = outPath / "css"
val fontsPath = outPath / "fonts"
```

Scala.js' output path can be remapped using:

```
.settings(
  ...
  artifactPath in (Compile, packageScalaJSLauncher) :=
    jsPath / "launcher.js"
, artifactPath in (Compile, fastOptJS) :=
    jsPath / "application.js"
)
```

Make sure to also add the following three paths to your `.gitignore`:

```
web/css/
web/js/
web/fonts/
```

4.6 sbt-web

Many popular web libraries are published to Maven Central as regular `.jar` files, so-called [WebJars](#). See the [official Scala.js documentation](#) on how to depend on these.

`sbt-web` is an sbt plug-in to manage these WebJars and to produce web artifacts as part of the build process. To enable `sbt-web`, add two imports:

```
import com.typesafe.sbt.web.SbtWeb
import com.typesafe.sbt.web.Import._
```

And enable the plug-in:

```
.enablePlugins(SbtWeb)
```

For example, to download the [Sass version of the Bootstrap bindings](#) as well as Font-Awesome, add these two lines to `libraryDependencies`:

```
libraryDependencies += Seq(
  ...
  "org.webjars" % "bootstrap-sass" % "3.3.1"
, "org.webjars" % "font-awesome" % "4.3.0-1"
)
```

Note: `sbt-web` is not necessary to use Bootstrap or Font-Awesome, albeit it facilitates the customisation and upgrading of web dependencies. The chapter ‘Bindings’ explains how to use a CDN instead.

4.6.1 Sass

Sass is a CSS dialect with useful extensions. One of its strengths is that you can modularise your stylesheet and store it in separate files. Since Bootstrap is available as Sass, the [sbt-sass plug-in](#) for `sbt-web` lets you create one monolithic, minified CSS file for your whole application. You may find that the widgets Bootstrap provides are not sufficient for your purposes. Using Sass, you would not end up with additional CSS files that need to be included in your `application.html`, which in turn will increase load times.

Assuming that you want to use Bootstrap and Font-Awesome in your application, create the directory `src/main/assets/` with the file `application.scss` containing:

```
$icon-font-path: "../fonts/";
@import "lib/bootstrap-sass/stylesheets/bootstrap.scss";

$fa-font-path: "../fonts/";
@import "lib/font-awesome/scss/font-awesome.scss";
```

Then, add to your `plugins.sbt`:

```
resolvers += Resolver.url(
  "GitHub repository"
, url("http://shaggyyeti.github.io/releases")
)(Resolver.ivyStylePatterns)

addSbtPlugin("default" % "sbt-sass" % "0.1.9")
```

And configure the output path of the produced CSS files in your `Build.scala`:

```
resourceManaged in sass in Assets := cssPath
```

Finally, add to your `.gitignore`:

```
.sass-cache/
```

`sbt-sass` requires that the official Sass compiler is installed on your system.

4.6.2 Font-Awesome

In order to automatically copy the Font-Awesome files to your configured path `fontsPath`, you can define a `sbt-web` task:

```
val copyFontsTask = {
  val webJars = WebKeys.webJarsDirectory in Assets
  webJars.map { path =>
    val fonts = path / "lib" / "font-awesome" / "fonts"
    fonts.listFiles().map { src =>
      val tgt = fontsPath / src.getName
      IO.copyFile(src, tgt)
      tgt
    }.toSeq
  }
}.dependsOn(WebKeys.webJars in Assets)
```

And register it via:

```
sourceGenerators in Assets <+= copyFontsTask
```

4.6.3 Artifacts

When you issue the `sbt` task `assets`, `sbt-web` will generate your web artifacts, like CSS files.

4.7 Code sharing

Scala.js provides a simple infrastructure to having separate sub-projects for JavaScript and JVM sources, which can share code. This is quite common for client-server applications which could have a common protocol specified in Scala code. You can work on your entire project in the IDE and easily jump between client and server code.

Such projects are called *cross-projects* in Scala.js. You can find more information in the [official documentation](#).

```
import org.scalajs.sbtplugin.cross.CrossProject

object Build extends sbt.Build {
  lazy val crossProject =
    CrossProject(
      "server", "client", file("."), CrossType.Full
    )
  .settings(
    /* Shared settings */
  )
  .jsSettings(
```

```
    /* Scala.js settings */  
  )  
  .jvmSettings(  
    /* JVM settings */  
  )  
  
  lazy val js = crossProject.js  
  lazy val jvm = crossProject.jvm  
}
```

You will also need to move your current `src/` folder to `js/`. The JVM project goes underneath `jvm/src/main/scala/` and the shared source files underneath `shared/src/main/scala/`.

4.8 Colours

Colours can be activated in sbt and Scala by setting two environment variables:

```
export JAVA_OPTS="$JAVA_OPTS -Dscala.color"  
export SBT_OPTS="$JAVA_OPTS"
```

ROUTER

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

The router observes the fragment identifier of the browser URL. For example, in `application.html#/page` the part after the hash mark denotes the fragment identifier, that is `/page`. The router is initialised with a set of known routes. A fallback route may also be specified.

5.1 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 set of enabled routes. It should also contain the `notFound` route. Otherwise, the page could not be displayed when `#/404` is loaded.

5.2 Routes

To construct a new route, pass the path and its reference to `Route()`. Pages may be overloaded with different paths as above with `test` and `test2`.

A path consists of *parts* which are separated by slashes. For instance, the `test` route above has two parts: `test` and `:param`. A part beginning with a

colon is a *placeholder*. Its purpose is to match the respective value in the fragment identifier and to bind it to the placeholder name. Note that a placeholder always refers to the whole part.

A route can be *instantiated* by calling it, setting all of its placeholders:

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

// One parameter
val route = Main.test("param", "value")

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

// Redirect to `route`
route.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"))
  }
}
```

5.2.1 Design decisions

Due to its limitations, the router could be efficiently implemented. Matching string-only parts in routes allows for better reasoning than regular expressions. When the router is constructed, it sorts all routes by their length and checks whether there are any 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()`.

5.3 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.

WIDGETS

A widget is a type-safe abstraction for an element displayed by the browser. The entire page layout is described using widgets. Thus, widget instantiations can be nested. Furthermore, custom widgets can be defined for better code reuse. A custom widget is usually composed of other widgets, changing their attributes such as CSS tags.

Instead of accessing DOM elements using `getElementById()`, a widget doesn't have any ID by default. Instead, it maintains a reference to the DOM element. This way, widgets that may have the same ID cannot collide and no ill-defined type-casts may occur.

Mutation methods on a widget return the instance. This allows to arbitrarily nest widgets and change their attributes by chaining method calls, without the need to store the widget in a local variable.

6.1 HTML

Widok provides widgets for many HTML elements. The bindings have a more intuitive naming than their HTML counterparts, although aliases were defined, too. The module the HTML widgets reside in is `org.widok.bindings.HTML`. If your project doesn't define any conflicting types, it is safe to import the whole contents into the namespace.

Alias	Widget	Notes
section	Section	
header	Header	
footer	Footer	
nav	Navigation	
h1	Heading.Level1	
h2	Heading.Level2	
h3	Heading.Level3	
h4	Heading.Level4	
h5	Heading.Level5	
h6	Heading.Level6	
p	Paragraph	

Alias	Widget	Notes
b	Text.Bold	
strong	Text.Bold	
i	Text.Italic	
small	Text.Small	
br	LineBreak	
hr	HorizontalLine	
div	Container.Generic	
span	Container.Inline	
raw	Raw	span with innerHTML
form	Form	
button	Button	
label	Label	
a	Anchor	
img	Image	
radio	Input.Radio	input with type="radio"
checkbox	Input.Checkbox	input with type="checkbox"
file	Input.File	input with type="file"
select	Input.Select	input with type="select"
text	Input.Text	input with type="text"
textarea	Input.Textarea	
password	Input.Password	input with type="password"
number	Input.Number	input with type="number"
option	Input.Select.Option	
ul	List.Unordered	
ol	List.Ordered	
li	List.Item	
table	Table	
thead	Table.Head	
th	Table.HeadColumn	
tbody	Table.Body	
tr	Table.Row	
td	Table.Column	
cursor	Cursor	

6.1.1 Aliases

By importing `org.widok.html._` you can use regular HTML tags instead of the more verbose notations.

6.2 Usage

A widget inherits from the type `Widget`. Widgets are implemented as `case classes` and can therefore be used like regular function calls. The simplest widget is `Raw()` which allows to render HTML markup:

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

This is equivalent to:

```
val widget = Text.Bold(
  Text.Italic("Text")
)
```

Most widgets take children. If this is the case, child widgets are passed per convention with the constructor. Widget parameters are set using chainable method calls:

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

6.3 Writing custom widgets

Widgets should be designed with type-safety in mind. For example, the only children `List.Unordered()` accepts are instances of `List.Item`. When creating custom widgets, think of a class hierarchy which closely resembles the intended nesting. This will allow to catch usage errors during compile-time.

A custom widget may be defined as follows:

```
case class Panel(contents: View*) extends Widget[Panel] {
  val rendered = DOM.createElement("div", contents)
  css("panel")
  css("panel-default")
}
```

This corresponds to:

```
<div class="panel panel-default">
  ... rendered children ...
</div>
```

If a custom widget doesn't need to be used as a type, it is sufficient to define it as a function:

```
def Panel(contents: View*) = Container.Generic(contents: _*)
  .css("panel")
  .css("panel-default")
```

6.4 Binding to events

A widget provides functionality to interact with the DOM. Methods with the prefix `on*` exist for all events and take a callback.

To listen to JavaScript's `onclick` and `ondblclick` events of a button, write:

```
Button("Click")
  .onClick(e => println("Click: " + e.timeStamp))
  .onDoubleClick(e => println("Double click: " + e.timeStamp))
```

All DOM events are published as channels. A channel produces data which is passed on to its subscribers. The above is a shortcut for:

```
val btn = Button("Click")
btn.click.attach(...)
btn.doubleClick.attach(...)
```

This allows for an event to 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 where the user updates the quantity of a certain product. At the same time the header needs to get updated with the newly calculated price. Making the DOM events available as streams widens the range of possibilities. As `click` is a stream of events, we could decide to take into account only the first event:

```
btn.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(btn.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, the widget is hidden as soon as we click the button.

Data propagation mechanisms are explained in more detail in the next chapter 'Reactive programming'.

6.5 Composed widgets

Widok provides a couple of composed widgets without external rendering dependencies. They are defined in the package `org.widok.widgets`:

- **LoremIpsum**: Prints Lorem Ipsum as a paragraph
- **Placeholder**: Generates placeholder images on-the-fly

6.6 Implicits

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

```
def view() = HTML.Paragraph("Litwo! Ojczyzno moja!")
```

Instead of:

```
def view() = Inline(HTML.Paragraph("Litwo! Ojczyzno moja!"))
```

Another implicit is evaluated here, which converts the string into a widget. There are also implicits to render buffers and channels.

REACTIVE PROGRAMMING

7.1 Motivation

User interfaces are heavily data-driven. Values do not only need to be displayed once, but continuously modified as the user interacts with the interface. Interactivity requires data dependencies which ultimately lead to deeply intertwined code. Imperative code in particular is prone to this shortcoming since dependencies are hard to express. As web applications are increasingly more interactive, a flow-driven approach is desirable. Focussing on flows, the essence of the program will be to specify the data dependencies and how values propagate to the user interface and back.

To tackle this issue, Widok follows a reactive approach. Consider an application to visualise stock market data. You are listening to a real-time stream producing values. Then, you want to display only the most current stock price to the user. This is solved by creating a ‘container’ which is bound to a DOM node. Whenever you feed a new stock price to it, an atomic update takes place in the browser, only changing the value of the associated DOM node.

Another example is a monitoring service which allows you to control on-the-fly the log level of a web application. A select box will list all possible log levels, like `debug` or `critical`. When the page is first loaded, it obtains the current log level from the server. Changing its value, however, must back-propagate and send the selection to the server. All other clients that are connected are notified of the change as well.

For a simple application that illustrates client-side data propagation, see our [TodoMVC implementation](#).

7.2 Concepts

Reactive programming is a paradigm that focuses on:

- a) propagation of data, specifically changes, and
- b) data flow.

Concretely, a data structure is said to be *reactive* (or *streaming*) if it models its state as streams. It does this by defining change objects (*deltas*) and mapping its

operations onto these. The published stream is read-only and can be subscribed. If the stream does not have any subscribers, the state would not get persisted and is lost.

Example: A reactive list implementation could map all its operations like `clear()` or `insertAfter()` on the two delta types `Insert` and `Delete`. A subscriber can interpret the deltas and persist the computed list in an in-memory buffer.

Another property of a reactive data structure is that it does not only stream deltas, but also state observations. Sticking to the reactive list example, the deltas could allow streaming observations on the list's inherent properties — one being the length, another the existence of a certain element, i.e. `contains(value)`.

Finally, a *mutable* reactive data structure is an extension with the sole difference that it maintains an internal state which always represents the computed result after a delta was received. This is a hybrid solution bridging mutable object-oriented objects with reactive data structures. The mutable variant of our reactive list could send its current state when a subscriber is registering. This ultimately leads to better legibility of code as subscribers can register at any point without caring whether the expected data has been propagated already. The second reason is that otherwise we would need multiple instances of mutual objects that interpret the deltas. This is often undesired as having multiple such instances incurs a memory overhead.

To recap, a reactive data structure has four layers:

- **State:** interpretation of the delta stream and “converting” it into a mutable object
- **Mutation operations:** functions to produce deltas on the stream¹
- **Polling operations:** blocking functions to query the state
- **Streaming operations:** publish the state changes as a stream

Obviously, the first three layers are the very foundation of object-orientation. It is different in that a) modifications are encoded as deltas and b) there are streaming operations.

For now we just covered the first component of reactive programming: data propagation. The second cornerstone, data flow, is equally important, though. Streams describe data flow in terms of dependencies. Considering you want to plot a line as a graph using the formula $y = mx + b$ and the user provides the values for m and b , then you would wrap these inputs in channels and express the dependencies using combinators²:

```
val m = Opt[Int]()
val b = Opt[Int]()
```

```
// Produces when user provided `m` and `b`
```

¹These functions do not access the state in any way.

²The types in the code only serve illustration purposes

```

val mAndB: ReadChannel[(Int, Int)] = m.combine(b)

// Function channel to calculate `y` for current input
val y: ReadChannel[Int => Int] =
  mAndB.map { case (m, b) =>
    (x: Int) => m * x + b
  }

```

The user could listen to `y` and whenever it receives a new function, it can just call it for all the `x` in the interval of the shown graph. The example shows that messages in streams are not bound to data objects and even immutable functions could be passed around.

The data propagation is illustrated by the following diagram:

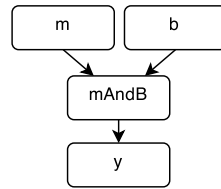


FIGURE 7.1 Change propagation for $y = mx + b$

As soon as the user inserts a value for `m` as well as `b`, `mAndB` will produce a tuple. Then, `y` computes the final function.

How channels work in detail is explained in the following sections. This example should only give an intuition of the fundamental concepts and how data dependencies are expressed.

7.3 Requirements

The term “stream” was used several times. This term is polysemous and requires further explanation. In reactive programming there are different types of streams with severe semantic differences.

`Rx` (Reactive Extensions) is a contract designed by Microsoft which calls these streams *observables* and defines rules how to properly interact with these. An observable can be subscribed to with an *observer* which has one function for the next element and two auxiliary ones for handling errors and the completion of the stream. Furthermore, observables are subdivided into *cold* and *hot* observables³:

³Source: leecampbell.blogspot.de (4th February 2015)

- **Cold observable:** Streams that are passive and start publishing on request
- **Hot observable:** Streams that are active and publish regardless of subscriptions

There are extensions to Rx which introduce back-pressure⁴ to deal with streams that are producing values too fast. This may not be confused with back-propagation which describes those streams where the subscribers could propagate values back to the producer.

This illustrates the diversity of streams. Due to the nature of Widok, streams had to be implemented differently from the outlined ones. Some of the requirements were:

- lightweight design
- support for n-way binding
- usable as the basis for reactive data structures
- provide functionality for resource management
- require little boilerplate to define new operations

To better differentiate from the established reactive frameworks, a less biased term than *observable* was called for and the reactive streams are therefore called *channels* in Widok. The requirements have been implemented as follows: A subscriber is just a function pointer (wrapped around a small object). A channel can have an unlimited number of children whereas each of the child channels knows their parent. A function for flushing the content of a channel upon subscription can be freely defined during instantiation⁵. When a channel is destroyed, so are its children. Error handling is not part of the implementation. Similarly, no back-pressure is performed, but back-propagation is implemented for some selected operations like `biMap()`.

For client-side web development only a small percentage of the interaction with streams require the features observables provide and this does not justify a more complex overall design. It is possible to use a full-fledged model like Rx or Monifu for just those areas of the application where necessary by redirecting (piping) the channel output.

7.4 Implementation

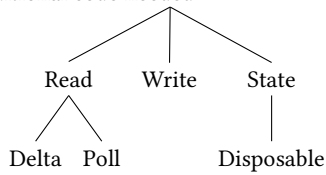
This section explains how reactive data structures are implemented in Widok. The design decisions will be beneficial for you to better understand the API and to design your own reactive data structures.

To leverage the capabilities of Scala's type system, we decided to separate the logic into distinct traits. Each data structure defines six traits which, when

⁴For instance, `Monifu` implements this feature.

⁵This function is called by `attach()` and produces multiple values which is necessary for some reactive data structures like lists.

combined using the Cake pattern, yield a mutable reactive object without any additional code needed:



For a hypothetical reactive data structure X you would define:

```

object X {
  /* Define delta change type */
}

/* Read/write access to state */
trait StateX[T] extends Disposable {
  /* This could be any kind of mutable storage */
  val state: Storage[T] = ...
  /* Channel needed by the other traits */
  val changes: Channel[X.Delta[T]] = ...
  /* Listen to `changes` and persist these in `state` */
  changes.attach { ... }
  /* Free resources */
  def dispose() { changes.dispose() }
}

/* The name may suggest otherwise, but it does not have any access
 * to the state; it only produces delta objects
 */
trait WriteX[T] {
  val changes: WriteChannel[X.Delta[T]]
  /* Also define operations to generate delta change objects */
}

trait DeltaX[T] {
  val changes: ReadChannel[X.Delta[T]]
  /* Also define streaming operations that listen to changes
 * and process these
 */
}

trait PollX[T] {
  val changes: ReadChannel[X.Delta[T]]
  /* Only read-only access is permitted here */
  val state: Storage[T]
  /* Also define streaming operations that need the state */
}

```

```

}

trait ReadX[T] extends DeltaX[T] with PollX[T]

case class X[T]()
  extends ReadX[T]
  with WriteX[T]
  with StateX[T]

```

A call to `X()` now yields a mutable reactive instance of our newly defined data structure.

It would have been possible to implement `X` as a single class, but the chosen approach offers more flexibility. Each of the traits are exchangeable. There are more possibilities for object instantiations. For example, often a change stream is already available. In this case, `DeltaX[T]` could be instantiated with a custom value for `changes`. The caller can decide whether it needs any of the operations that `PollX` defines. Depending on this decision it will either buffer the data or not. This ultimately leads to a more memory-efficient design as the responsibility of memory allocation is often shifted to the caller. It is in some way similar to what Python allows with its `yield` expression.

The delta trait has a read-only version of the change stream. It may define operations that apply transformations directly on the stream without building any complex intermediate results. A prominent example would be the higher-order function `map()`. As `map()` works on a per-element basis and does not need any access to the state, it can be implemented efficiently. As a consequence, this allows for chaining: `list.map(f).map(g).buffer` would compute the final list at the very end with the `buffer` call⁶.

Another motivating reason for this design is precisely the immutability of delta objects. The stream could be forwarded directly to the client which may render the elements in the browser on-the-fly. A similar use case would be persistence, for example in an asynchronous database.

Scala's type refinements for traits come in useful. `X` takes `changes` from `StateX`. It points to the same memory address in `WriteX` and `DeltaX` even though they are declared with different types. This is because `Channel` inherits both from `WriteChannel` and `ReadChannel`.

The type-safety has an enormous benefit: A function can use a mutable stream internally, but returning the stream with writing capabilities would lead to unpredictable results. If the caller accidentally writes to this stream, this operation will succeed and in the worst case other subscribers receive the messages as well. As `X` inherits from `ReadX`, the function can be more explicit and revoke some of its capabilities simply by returning `ReadX[T]`. Similarly, if the caller *should* get writing capabilities and no read capabilities, this can be made explicit as well. This will make it trivial to find bugs related to reading and writing capabilities of streams directly during compile-time. And it makes in-

⁶This is largely inspired by Scala's `SeqView`.

terfaces more intelligible as a more specific type reduces the semantic space of a function.

The third advantage is correctness: With the functionality separated into different traits, the proper behaviour can be ensured using property-based testing. Rules for the generation of delta objects could be defined⁷. This stream is then used in `StateX` and all other traits can be tested whether they behave as expected. Presently, a very basic approach for property-based testing is implemented, but future versions will explore better ways to achieve a higher coverage.

A variety of generally applicable reactive operations were specified as traits in `org.widok.reactive`. They can be seen as a contract and a reactive data structure should strive to implement as many as possible of these. Depending on conceptual differences, not every operation can be defined on a data structure, though. As the signatures are given, this ensures that all data structures use the operations consistently. Each of the traits group functions that are similar in their behaviour. Furthermore, the traits are combined into sub-packages which follow the properties mentioned at the beginning of the chapter, namely `org.widok.reactive.{mutate, poll, stream}`.

To summarise, for a reactive data structure it is necessary to declare several traits with the following capabilities:

TABLE 7.1 Traits and layers of a reactive data structure

	State	Mutation	Polling	Streaming
<code>Delta</code>	no	no	no	yes
<code>Poll</code>	no	no	yes	yes ⁸
<code>Read</code>	no	no	yes	yes
<code>Write</code>	no	yes	no	no
<code>State</code>	yes	no	no	no

7.5 Reactive data structures

Widok currently implements four reactive data structures:

- **Channels:** single values like `T`
- **Buffers:** lists like `Seq[T]`
- **Dictionaries:** maps like `Map[A, B]`
- **Sets:** reactive `Set[T]`

⁷For example, a `Delta.Clear` may only be generated after `Delta.Insert`.

⁸This is a practical decision. The `Poll` trait has direct access to the state. Thus, certain streaming operations can be implemented more efficiently. This should be avoided though as a delta stream would need to be persisted first in order for the `Poll` trait to be applicable.

7.6 Channels

A channel models continuous values as a stream. It serves as a multiplexer for typed messages that consist of immutable values. Messages sent to the channel get propagated to the observers that have been attached to the channel — in the same order as they were added. It is possible to operate on channels with higher-order functions such as `map()`, `filter()` or `take()`. These methods may be chained, such that every produced values is propagated down the observer chain.

Widok differentiates between two top-level channel types:

- **Channel**: corresponds to a reactive `T`
- **Partial channel**: corresponds to a reactive `Option[T]`

There are four channel implementations:

- **Channel**: stream that does not persist its values
- **Var**: variable stream; its value is always defined and has an initial value⁹
- **LazyVar**: stream for lazily evaluated variables
- **PtrVar**: stream for generic events¹⁰

Partial channels model optional values:

- **PartialChannel**: base type
- **Opt**: stream that has two states, either *defined with a value* or *undefined*

Note: `Opt[T]` is merely a convenience type and `Var[Option[T]]` could be used, too.

7.6.1 Operations

Here is a simple example for a channel that receives integers. We register an observer which prints all values 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`.

The return values of operations are channels, therefore chaining is possible. Channels can be used to express data dependencies:

⁹In Rx terms, `Var` would correspond to a *cold observer* as attaching to it will flush its current value. This is different from `Channel` which loses its messages when there are no subscribers.

¹⁰It can be used to create delta channels from DOM variables by binding to the corresponding events that triggered by the value changes. For an example see `Node.click`.


```

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

```

Use the method `distinct` to produce a value if it is the first or different from the previous one. A use case is to perform time-consuming operations such as performing HTTP requests only once for the same user input:

```

ch.distinct.attach { query =>
  // perform HTTP request
}

```

It must be noted that streaming operations have different semantics than their non-reactive counterparts. For brevity, only certain combinators are covered by the manual. For the rest, please refer to the ScalaDoc documentation.

7.6.2 State channels

For better performance, `Channel` does not cache the produced values. Some operations cannot be implemented without access to the current value, though. And often it is necessary to poll the current value. For these reasons *state channels* such as `Var` or `Opt` were introduced. The following example visualises the different behaviours:

```

val ch = Var(42)
ch.attach(println) // prints 42

val ch2 = Channel[Int]()
ch2 := 42 // Value is lost as ch2 does not have any observers
ch2.attach(println)

```

`update()` is an operation that requires that the produced values are persisted. `update()` takes a function which modifies the current value:

```

val ch = Var(2)
ch.attach(println)
ch.update(_ + 1) // produces 3

```

A partially-defined channel (`Opt`) is constructed as follows:

```

val x = Opt[Int]()
x := 42

```

Alternatively, a default value may be passed:

```

val x = Opt(42)

```

A state channel provides all the methods a channel does. `Var [T]` and `Opt [T]` can be obtained from any existing `ReadChannel [T]` using the method `cache`:

```
val chOpt = ch.cache      // Opt[Int]
val chVar = ch.cache(42) // Var[Int]
```

`chOpt` is undefined as long as no value was produced on `ch`. `chVar` will be initialised with 42 and the value is overridden with the first produced value on `ch`.

`biMap()` allows to implement a bi-directional map, i.e. a stream with back-propagation:

```
val map    = Map(1 -> "one", 2 -> "two", 3 -> "three")
val id     = Var(2)
val idMap  = id.biMap(
  (id: Int)    => map(id)
  , (str: String) => map.find(_._2 == str).get._1)
id    .attach(x => println("id   : " + x))
idMap.attach(x => println("idMap: " + x))
idMap := "three"
```

The output is:

```
id   : 2
idMap: two
id   : 3
idMap: three
```

`biMap()` can be used to implement a lens as a channel. The following example defines a lens for the field `b`. It has a back channel that composes a new object with the changed field value.

```
case class Test(a: Int, b: Int)
val test = Var(Test(1, 2))
val lens = test.biMap(_._b, (x: Int) => test.get.copy(b = x))
test.attach(println)
lens := 42 // produces Test(1, 42)
```

A `LazyVar` evaluates its argument lazily. In the following example, it points to a mutable variable:

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

7.6.3 Call semantics

Functions passed to higher-order operations are evaluated on-demand:

```
val ch = Var(42).map(i => { println(i); i + 1 })
ch.attach(_ => ()) // prints 42
ch.attach(_ => ()) // prints 42
```

The value of a state channel gets propagated to a child when it requests the value (`flush()`). In the example, `Var` delays the propagation of the initial value 42 until the first `attach()` call. `attach()` goes up the channel chain and triggers the flush on each channel. In other words, `map(f)` merely registers an observer, but doesn't call `f` right away. `f` is called each time when any of its direct or indirect children uses `attach()`.

This reduces the memory usage and complexity of the channel implementation as no caching needs to be performed. On the other hand, you may want to perform on-site caching of the results of `f`, especially if the function is side-effecting.

The current value of a state channel may be read at any time using `.get` (if available) or `flush()`.

There are operations that maintain state for all observers. For example, `skip(n)` counts the number of produced values¹¹. As soon as `n` is exceeded, all subsequent values are passed on. The initial `attach()` calls ignore the first value (42), but deal with all values after that:

```
val ch = Var(42)
val dch = ch.drop(1)
dch.attach(println)
dch.attach(println)
ch := 23 // produces 23 twice
```

7.7 Buffers

Buffers are reactive lists. State changes such as row additions, updates or removals are encoded as delta objects. This allows to reflect these changes directly in the DOM, without having to re-render the entire list. `Buffer[T]` is therefore more efficient than `Channel[Seq[T]]` when dealing with list changes.

The following example creates a buffer with three initial rows, observes the size¹² and then adds another row:

```
val buf = Buffer(1, 2, 3)
buf.size.attach(println) // Prints 3
buf += 4 // Inserts row 4, prints 4
```

All polling methods have a dollar sign as suffix `$`:

¹¹`n` must be greater than 0.

¹²`size` returns a `ReadChannel[Int]`.

```
val buf = Buffer(1, 2, 3)
println(buf.size$) // Prints 3
```

An example of using `removeAll()`:

```
val buf = Buffer(3, 4, 5)
val mod2 = buf.filter$_ % 2 == 0)

buf.removeAll(mod2.get)
```

Note: `Buffer` will identify rows by their value if the row type is a `case class`. In this case, operations like `insertAfter()` or `remove()` will always refer to the first occurrence. This is often not desired. An alternative would be to define a `class` instead or to wrap the values in a `Ref[_]` object:

```
val todos = Buffer[Ref[Todo]]()
ul(
  todos.map { case tr @ Ref(t) =>
    li(
      // Access field `completed`
      checkbox().bind(t.completed)

      // remove() requires reference
      , button().onClick(_ => todos.remove(tr))
    )
  }
)
```

The value of a `Ref[_]` can be obtained by calling `get`. However, it is more convenient to do pattern matching as in the example.

You can observe the delta objects produced by a buffer:

```
val buf = Buffer(1, 2, 3)
buf.changes.attach(println)
buf += 4
buf.clear()
```

This prints:

```
Insert(Last(),1)
Insert(Last(),2)
Insert(Last(),3)
Insert(Last(),4)
Clear()
```

All streaming operations that a buffer provides are implemented in terms of the `changes` channel.

7.8 Dictionaries

Dictionaries are unordered maps from A to B. `Widok` abbreviates the type as `Dict`.

7.9 Sets

Reactive sets are implemented as `BufSet`¹³.

7.10 Binding to Widgets

Reactive data structures interact with user interfaces. These data structures are usually set up before the widgets, so that they can be referenced during the widget initialisation. The most common use case is binding channels to DOM nodes:

```
val name = Channel[String]()
def view() = h1("Hello ", name)
```

This example shows one-way binding, i.e. uni-directional communication. `name` is converted into a widget, which observes the values produced on `name` and updates the DOM node with every change. This is realised by an implicit and translates to `span().subscribe(name)`.

Another implicit is provided for widget channels, so you can use `map()` on any channel to create a widget stream. The widgets are rendered automatically. If the widget type stays the same and it provides a `subscribe()` method, use it instead.

On form fields you will need to call `subscribe()` by yourself:

```
val name = Channel[String]()
def view() = text().subscribe(name)
```

Two-way binding is achieved by using the method `bind()` instead of `subscribe()`. The only difference is that changes are back-propagated. This lets you define multiple widgets which listen to the same channel and synchronise their values:

```
val ch = Var("Hello world")
def view() = Inline(
  Input.Text().bind(ch)
, Input.Text().bindEnter(ch)
)
```

This creates two text fields. When the page is loaded, both have the same content: “Hello world”. When the user changes the content of the first field, the second text field is updated on-the-fly. The second field requires an enter press before the change gets propagated to the first text field.

¹³This name was chosen as `Set` would have collided with Scala’s implementation.

Each widget has methods to control its attributes either with static values or channels. For example, to set the CSS tag of a widget use `widget.css("tag1", "tag2")`. This method is overloaded and you could also pass a `ReadChannel[Seq[String]]`.

Passing channels is useful specifically for toggling CSS tags with `cssState()`. It sets CSS tags only when the expected channel produces `true`, otherwise it unsets the tags:

```
widget.cssState(editing, "editing", "change")
```

Other useful functions are `show()` and `visible()`. The former sets the CSS property `display` to `none`, while the latter sets `visibility` to `hidden` to hide a widget.

As reactive data structures provide streaming operations that return channels, these can be used in widgets. Consider the method `isEmpty` that is defined on buffers. You could show a `span` depending on whether the list is empty or not:

```
val buf = Buffer[Int]()

def view() = Inline(
  span("The list is empty.")
    .show(agg.isEmpty)
, span("The list is not empty.")
    .show(agg.nonEmpty)
, button().onClick(_ => buf += 42)
, button().onClick(_ => buf.clear())
)
```

7.11 Tests

The proper functioning of each operation is backed by [test cases](#). These provide complementary documentation.

BINDINGS

This chapter deals with third-party CSS frameworks for which Widok provides typed bindings.

8.1 Bootstrap

Bootstrap is a “framework for developing responsive, mobile first projects on the web.” See the [project page](#) for more information.

To use the bindings, it may be desirable to import its entire namespace:

```
import org.widok.bindings.Bootstrap._
```

Bootstrap’s components closely resemble their HTML counterparts. For example:

```
<button type="button" class="btn btn-default">
  <span class="glyphicon glyphicon-align-left"></span>
</button>
```

This translates to:

```
Button(Glyphicon.AlignLeft())
```

Bootstrap widgets expect a list of arguments which denotes child widgets. The configuration can be controlled by usual method calls on the widget. If a widget conceptually doesn’t have any children, then its arguments are used for the configuration instead.

8.1.1 External stylesheet

For the bindings to work without sbt-web, add the latest Bootstrap stylesheet to the head tag of your `application.html` file. You can either keep a local copy of the stylesheet or use a CDN:

```
<link
  rel="stylesheet"
  href="https://maxcdn.bootstrapcdn.com/
bootstrap/3.3.2/css/bootstrap.min.css">
```

Please keep in mind that the pre-built stylesheet comes with certain restrictions, like the font path being hard-coded.

8.1.2 Label

Every widget is equipped with a method `label(value: Style)` that allows attaching a Bootstrap label like `label-info` to it:

```
span("Text").label(Label.Info)
```

`TextContainer(styles: Style*)` generates a container with a list of styles.

8.1.3 Glyphicons

Glyphicons are simple function calls, for example: `Glyphicon.User()`. All Bootstrap glyphicons are supported, although the naming was changed to camel-case.

8.1.4 Forms

Forms can be validated on-the-fly. For each field a custom validator may be written. `validator.errors()` will render the textual error. Instead of showing the error underneath a field, this call can be placed anywhere, for instance to centralise all errors. `validate()` is defined on every widget and sets the `has-error` CSS tag if a field is invalid. The initial validation is triggered when the user presses the submit button. `validator.check()` will perform the first validation and return `true` if all fields are valid. If at least one input field was invalid, the submit button is kept disabled as long as the input stays wrong.

```
val username      = Var("")
val displayName   = Var("")

def validateNonEmpty(value: String) =
  if (value.trim.isEmpty) Some("Field cannot be empty")
  else None

implicit val validator = Validator(
  Validation(username, validateNonEmpty)
, Validation(displayName, validateNonEmpty)
)

Container(
  FormGroup(
    InputGroup(
      InputGroup.Addon(Glyphicon.Globe())
    , Input.Text()
```



```

        .placeholder("Display name")
        .size(Size.Large)
        .tabIndex(1)
        .bind(displayName)
    )
    , validator.errors(displayName)
).validate(displayName)

, FormGroup(
    InputGroup(
        InputGroup.Addon(Glyphicon.User())
    , Input.Text()
        .placeholder("Username")
        .size(Size.Large)
        .tabIndex(2)
        .bind(username)
    )
    , validator.errors(username)
).validate(username)
)

, Button("Submit").onClick { _ =>
    if (validator.check()) println("Ok")
}.enabled(validator.maySubmit)
)

```

Other widgets related to forms are:

- `HorizontalForm()`
- `FormGroup()`
- `ControlLabel()`
- `InputGroup()`
- `InputGroup.Addon()`
- `Input.Text()`
- `Input.Password()`
- `Input.Select()`
- `Button()`
- `Button.Group()`
- `Button.Toolbar()`
- `Checkbox()`

8.1.5 Layout

Layout-related widgets are:

- `Footer()`

- `Container()`
- `PageHeader()`
- `Lead()`
- `PullRight()`

8.1.6 Navigation

Example:

```
val tab1 = Navigation.Tab("Tab 1")
val tab2 = Navigation.Tab("Tab 2")
val currentTab = Var(tab1)
```

```
Navigation.renderTabs(Seq(tab1, tab2), currentTab)
```

8.1.7 Navigation bar

Example for the `NavigationBar` widget:

```
NavigationBar(
  Container(
    NavigationBar.Header(
      NavigationBar.Toggle()
      , NavigationBar.Brand("Brand name")
    )
    , NavigationBar.Collapse(
      NavigationBar.Elements(
        Item(a(Glyphicon.Dashboard(), " Page 1").url(Routes.page1()))
        , Item(a(Glyphicon.Font(), " Page 2").url(Routes.page2()))
        , NavigationBar.Right(
          NavigationBar.Navigation(
            NavigationBar.Form(
              FormGroup(
                InputGroup(Input.Text())
                , Button(Glyphicon.Search())
              ).role(Role.Search)
            )
          )
        )
      )
    )
  )
)
```

As probably more than one page is going to use the same header, you should create a trait for it. For example, you could define `CustomPage` with the header. Then, it only requires you to define the page title and body for every page.

8.1.8 Alert

Example:

```
Alert("No occurrences").style(Style.Danger)
```

8.1.9 Progress bar

Example:

```
val percentage = Var(0.1)
ProgressBar("Caption")
    .style(percentage.map(p => if (p < 0.5) Style.Warning else Style.Success)
    .progress(percentage)
```

8.1.10 Panel

Example:

```
Panel(
    Panel.Heading(Panel.Title3("Panel title"))
, Panel.Body("Panel text")
).style(Style.Danger)
```

8.1.11 Pagination

Example:

```
Pagination(
    Pagination.Item(a("«")).disabled(true)
, Pagination.Item(a("1")).active(true)
, Pagination.Item(a("2"))
, Pagination.Item(a("»"))
)
```

8.1.12 List groups

Example:

```
ListGroup(
    ListGroup.Item(a("Item 1")).active(true),
, ListGroup.Item(a("Item 2"))
, ListGroup.Item(a("Item 3"))
)
```

8.1.13 Grids

Example:

```
Grid.Row(
  Grid.Column(
    "Grid contents"
  ).column(Size.ExtraSmall, 6)
  .column(Size.Medium, 3)
)
```

8.1.14 Modal

It is most convenient to use the `ModalBuilder` to create modals. On the same page you can define several modals. For example:

```
val modal: ModalBuilder = ModalBuilder(
  Modal.Header(
    Modal.Close(modal.dismiss)
    , Modal.Title("Modal title")
  )
, Modal.Body("Modal body")
, Modal.Footer(
  Button("Submit").onClick(_ => modal.dismiss())
)
)

def body() = div(
  Button("Open").onClick(_ => modal.open())
, modal /* Each modal must be added to the body. It is hidden by default. */
)
```

8.1.15 Media

Example:

```
Media(
  Media.Left(Placeholder("cover", Placeholder.Size(150, 80)))
, Media.Body(
  Media.Heading("Heading")
, "Description"
)
)
```

8.1.16 Breadcrumb

```
Breadcrumb(
  Item(a("Item 1"))
)
```

```
, Item(a("Item 2")).active(true)
)
```

8.1.17 Table

To use a Bootstrap table, use `Table()` and `Table.Row()` which in contrast to `table()` and `tr()` provide Bootstrap-related styling options:

```
Table(
  head(
    tr(
      th("Date")
      , th("Quantity")
    )
  )
, tbody(
  Table.Row(td("01.01.2015"), td("23")).style(Style.Info)
  , Table.Row(td("02.01.2015"), td("42")).style(Style.Danger)
  )
)
```

8.1.18 Typeahead

Example:

```
val allMatches = Map(0 -> "First", 1 -> "Second", 2 -> "Third")
def matches(input: String): Seq[(Int, String)] =
  allMatches.filter { case (k, v) => v.startsWith(input) }.toSeq
def select(selection: Int) { println(s"Selection: $selection") }
```

```
Typeahead(Input.Text(), matches, select)
```

8.2 Font-Awesome

The Font-Awesome bindings include all icons in camel-case notation. For convenience, rename the object when you import it:

```
import org.widok.bindings.{FontAwesome => fa}
```

Using the `user` icon is as simple as writing:

```
fa.User()
```

This translates to:

```
<span class="fa fa-user"></span>
```


DEVELOPING

If you would like to participate or try out development releases, please read this chapter.

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. Remember that your project's Scala.js version must match the version Widok is built for.

9.3 Releases

The versioning scheme follows the format `releaseSeries.relativeVersion`. Thus, `v0.2.0` defines the version 0 of the release series 0.2. All versions within the same release series must be binary-compatible. If any of the dependencies (like Scala.js) are updated, the release series must be increased as well.

Widok releases are published to [Maven Central](#).

9.4 Manual

Since `v0.2`, the manual is stored in the same repository as the code. This enables you to commit code coupled with the corresponding documentation changes. At any time, the manual should always reflect the current state of the code base.

CHANGELOG

The changelog lists all major changes between releases. For more details, please see the [Git changelog](#).

10.1 Version 0.2

10.1.1 General work

- Relicensed as Apache v2.0
- Ported to Scala 2.11.5
- Ported to Scala.js v0.6.0
- Ported to scalajs-dom v0.8.0
- Uses [minitest](#) for the test cases
- [Shapeless](#) dependency dropped; the lens macros resulted in slower compilation and had insufficient IDE support. At the time of writing, Shapeless was not yet ported to Scala.js v0.6.0. `Var()` can be used as a drop-in replacement for more flexibility and better performance. As a result of dropping Shapeless, the Sonatype resolver is not needed anymore.
- There is a [Gitter channel](#) for conversations about Widok
- JVM support: Widok is now built for the JVM, allowing you to use the reactive library on the server, too. Upcoming versions will further focus on network transparency (see [#26](#)). In the future, the widget library will be usable under the JVM as well (see [#25](#)).
- The manual is now part of the code repository. Pandoc is used for compilation. Current targets are PDF, HTML and EPUB.

10.1.2 Event propagation

This version includes a complete redesign of the event propagation mechanisms. The previous implementation was merely a proof of concept and therefore had a couple of design issues. Changes include:

- More abstraction layers for better type-safety and modularity. The whole API now strictly distinguishes between `ReadChannels` and `WriteChannels`.

By looking at the types of a function, it is now more predictable what the passed channel is being used for.

- Improve naming: `Var` and `Buffer` as opposed to `CachedChannel` and `CachedAggregate`
- `Aggregates` were dropped. Instead, reactive data structures were introduced, which encode their changes as delta objects and are built on top of channels. Publishing changes as channels allows to persist the stream or send it directly to the client. Furthermore, change transformers could be written, for example one that cancels out common sequences like for better rendering performance:

```
Change.Insert(Position.Last(), element)
Change.Remove(element)
```

- Reactive data structures for buffers, dictionaries and sets were introduced.
- A reactive Rose tree implementation was added.
- Back propagation for aggregates was entirely removed. For sequences it offered little benefit. Previously, back propagation was only used for deletions. Abandoning this feature makes it easier to implement new combinators. For example, `filter()` became significantly shorter and better to comprehend.
- Partially defined streams were introduced (`Opt`)
- Created traits that fully specify the functionality of reactive combinators (see `Combinators.scala`). As most combinators are implemented by more than one class, this will ensure consistency in their usage. They could also be used outside of `Widok` as the basis for alternative implementations.
- Work on resource management has begun; `dispose()` can be called on channels to clear its subscription in the parent.
- Property-based testing for channels and aggregates (basic operations only)
- Cycle detection for channels. As cycles may be well-defined under certain circumstances, they can be ignored using `filterCycles` (see the [TodoMVC application](#) for an example)

10.1.3 Widgets

Some work also went into the widget subsystem:

- New implicits to render numerical values, buffers and sequences without type conversions

- DOM event listeners are now available on all widgets and made available as channels. This implies that more than one listener can be attached to the same event. Also, `on*()` short-hands were created for all events. This makes registering a click event listeners as simple as writing `.onClick(ev => ...)`.
- All DOM elements have short aliases which are equal to their HTML tag names. Instead of `Input.Text()`, you could now write `text()`. As these tag names may conflict in certain cases, an explicit import is necessary: `import org.widok.html._`
- The Bootstrap bindings were completely revised. No type casts are necessary anymore and most widgets are now `case classes` instead of functions. This makes the widgets easier in their usage. The bindings now cover a large percentage of the functionality Bootstrap provides.

Newly added widgets are:

- Bootstrap: Modal dialogue
- Bootstrap: Typeahead field
- Bootstrap: Form validation
- Bootstrap: Table
- Bootstrap: Pagination
- Bootstrap: Panel
- Bootstrap: Button group and toolbar
- Bootstrap: Breadcrumb
- Bootstrap: Media object
- HTML: File input
- HTML: Password field
- HTML: Select box
- HTML: Radio button
- HTML: HTML5 number input field
- Widget container `InLine()` (can be used when a `span()` would introduce undesired design glitches)
- Image placeholder
- Lorem ipsum

Code generators were introduced for higher reliability of the bindings. `sbt-web` is used internally to obtain external web dependencies. As part of the build process, Scala files are then created. Auto-generated bindings are provided for:

- Bootstrap's glyphsicons
- Font-Awesome

10.1.4 Routing

- In v0.1 the DOM nodes of all routes were initialised when the page loads. This may result in cycle errors. Now, a route change also reinitialises the

entire page.

- The method `destroy()` can be overridden to react on page changes, for example for manual resource management

10.1.5 Trivia

- The API changes significantly shrunk TodoMVC's file size from 3396 bytes (92 lines) to 3059 bytes (89 lines).
- Widok can be used to develop desktop applications with [NW.js](#) as shown by [poliglot-ui](#)

CHAPTER 11

SUPPORT

The official support channels are [Gitter](#) and [GitHub issues](#). If you have general questions, you may also send me an e-mail.