Korolev: Single Page Applications Framework

Table of Contents

Aleksey Fomkin <aleksey.fomkin@gmail.com>

1. Introduction

Not long ago we have entered the era of single-page applications. Some people say that we no longer need a server. They say that JavaScript applications can connect to DBMS directly. Fat clients. We disagree with this. This project is an attempt to solve the problems of the modern fat web.

Korolev runs a single-page application on the server side, keeping in the browser only a bridge to receive commands and send events. The page loads instantly and works fast because it does a minimal amount of computation. It's important that Korolev provides a unified environment for full stack development. Client and server are now combined into a single app without any REST protocol or something else in the middle.

Our project supports static page rendering so that search engines could index pages and users could view them immediately.

1.1. Principles

1. **Thin client.** Let's be honest, modern JavaScript applications are too greedy. Every JavaScript developer thinks that his page is special. But the user has a different opinion. He opens dozens of tabs. Each tab contains a ton of code, and this works very slowly. So we made our JavaScript bridge as lightweight as possible.
2. **Immutable and pure.** Really, we don't need mutability even on the frontend. Especially on the frontend. The majority of modern JavaScript frameworks try to be functional. So we do.
3. **Lack of dependencies.** Korolev does not impose to use cats, scalaz, Akka, or futures from the standard library. Use what you want. Korolev gives a way to integrate itself into any environment.

1.2. Quick start

You need SBT and JDK 9+ installed. Once installed, run g8 template.

```
$ sbt new fomkin/korolev.g8
$ cd mykorolevproject
$ sbt
> re-start
```

Ok, now you are ready to start hacking with Korolev.

2. Understanding Korolev

2.1. Template DSL

Korolev uses Levsha as DSL (domain specific language) for templating. The levsha DSL allows to declare DOM using Scala code. Take a look.

```
import levsha.dsl._
import html._

div(
  backgroundColor @= "yellow",
  input(`type` := "text")
)
```

This code fragment corresponds to HTML below.

```
<div style="background-color: yellow">
  <input type="text"></input>
</div>
```

As you can see there are three simple rules:

1. `tag()` declares tag
2. `attribute := "value"` declares attribute
3. `disabled`, `selected` is attributes too
4. `backgroundColor @= "black"` declares style

NOTE	Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes from JS (e.g. node.style.backgroundImage).

You can define custom tags and styles using `TagDef`, `AttrDef` and `StyleDef` API.

```
import levsha.dsl._
val myCustomElement = TagDef("myCustomElement")
val myDataAttr = AttrDef("data-attr")
val bottom = StyleDef("bottom")
```

If custom attribute and custom tag has same name, you can created mixed definition.

```
val awesome = TagDef with StyleDef {
  val ns = levsha.XmlNs.html
  val name = "awesome"
}
```

2.2. Device and Session

When a user opens Korolev application the first time, it sets a cookie named `device` with a unique identifier. After that, you can identify the user by `deviceId`.

A session is created when the user opens a page. Every session has a unique identifier `sessionId`.

Korolev has a separate *State* for every session.

2.3. State

State is the only source of data needed to render a page. If you have something you want to display, it should be stored in state. State is always immutable.

```
sealed trait MyState

case class Anonymous(
  deviceId: DeviceId) extends MyState

case class Authorized(
  deviceId: DeviceId,
  user: User) extends MyState

case class User(
  id: String,
  name: String,
  friends: Seq[String])
```

State stores in `StateStorage` and initializes in `StageLoader`.

```
StateLoader.forDeviceId { deviceId =>
  // This hypothetical service method returns Future[User]
  authorizationService.getUserByDeviceId(deviceId) map { user =>
    Authorized(deviceId, user)
  }
}
```

The only way to modify the State is by applying a **transition** (see below).

2.4. Render

Render is a map between state and view. In other words render is a function which takes state as an argument and returns a DOM. Korolev uses Levsha DSL to define templates using Scala code. Do not be afraid, you will quickly get used to it.

```
val render: State => Node = {
  case Anonymous(_) =>
    Html(
      body(
        form(
          input(placeholder := "Login"),
          input(placeholder := "Password"),
          button("Submit")
        )
      )
    )
  case Authorized(_, User(name, friends)) =>
    Html(
      body(
        div(s"Your name is $name. Your friends:"),
        ul(
          // Look at that. You just map data to view
          friends map { friend =>
            li(friend)
          }
        )
      )
    )
}
```

Render is called for each new state. It's important to understand that render produces a full DOM for every call. It does not mean that client receives a new page every time. Korolev makes a diff between current and latest DOM. Only changes between the two are sent to the client.

2.5. Transitions

Transition is a function that applies to the current state and produces a new state. Consider a transition which adds a new friend to the friends list.

```
val newFriend = "Karl Heinrich Marx"
// This hypothetical service method adds friend
// to the user friend list and returns Future[Unit]
userService.addFriend(user.id, newFriend) flatMap { _ =>
  access.transition { case state @ Authorized(_, user) =>
    state.copy(user = user.copy(user.friends :+ newFriend))
  }
}
```

`Transition` is a type alias for `S ⇒ S` where `S` is the type of your state.

2.6. Events

Every DOM event emitted which had been emitted by browser's `document` object, can be handled by Korolev. Let's consider click event.

```
case class MyState(i: String)

def onClick(access: Access) = {
  access.transition {
    case MyState(i) =>
      state.copy(i = i + 1)
  }
}

def renderAnonymous(state: MyState) = optimize {
  Html(
    body(
      state.i.toString,
      button("Increment",
        event("click")(onClick)
      )
    )
  )
}
```

When "Increment" button will be clicked, `i` will be incremented by 1. Let's consider a more complex situation. Do you remember render example? Did you have a feeling that something was missing? Let's complement it with events!

```
val loginField = elementId()
val passwordField = elementId()

def onSubmit(access: Access) = {
  for {
    login <- access.valueOf(loginField)
    password <- access.valueOf(passwordField)
    user <- authService.authorize(login, password)
    _ <- access.transition {
      case Anonymous(deviceId) =>
        Authorized(deviceId, user)
    }
  } yield ()
}

def renderAnonymous = optimize {
  Html(
    body(
      form(
        input(placeholder := "Login", loginField),
        input(placeholder := "Password", passwordField),
        button("Submit"),
        event("submit")(onSubmit)
      )
    )
  )
}
```

Event gives you an access to information from the client side. In this case, it it is values of the input fields. Also you can get data from event object using `access.eventData` function `.

Event handlers should return F[Unit] which will be implicitly converted to `EventResult`.

2.7. Stateful components

In the world of front-end development people often call every thing that has HTML-like markup a "component". Until version 0.6, Korolev didn't have components support (except WebComponents on client side). When we say "component" we mean something that has its own state. In Korolev case, components also dispatch events to other components up in the hierarchy or to the application on the top.

Worth to note that application is a component too, so you can consider Korolev's components system as a system composed of applications. Korolev components are not lightweight, so if you can implement a feature without components, do it so. Use components only when they are really necessary.

Components are independent. Every component has its own context. It opens up a way to write reusable code.

There are two ways to define a component: in functional and in object-oriented style. Let's take a look at functional style components.

```
val MyComponent = Component[Future, ComponentState, ComponentParameters,
EventType](initialState) { (context, parameters, state) =>

  import context._
  import levsha.dsl._
  import html._

  div(
    parameters.toString,
    state.toString,
    event("click") { access =>
      // Change component state here
      // using transition as usual.
    }
  )
}
```

Same component can be defined in object-oriented style.

```
object MyComponent extends Component[Future, ComponentState, ComponentParameters,
EventType](initialState) {

  import context._
  import levsha.dsl._
  import html._

  def render(parameters: ComponentParameters, state: ComponentState): Node = {
    div(
      parameters.toString,
      state.toString,
      event("click") {
        // Change component state here
      }
    )
  }
}
```

Let's use our component in the application.

```
Html(
  body(
    "Hello world!",
    MyComponent(parameters) { (access, _) =>
      // make transition on component event
      access.transition {
        case n => n + 1
      }
    },
    // If you don't want to handle event, use silent instance
    MyComponent.silent(parameters)
  )
)
```

See full example

2.8. Access element properties

In the scope of an event, you can access the element properties if an element was bound with `elementId`. You get the value as a string. If the value is an object (on client side) you will get JSON-string. You can parse it with any JSON-library you want.

```
val loginField = elementId()

...

event("submit") { access =>
  access.property(loginField, "value") flatMap {
    access.transition { ... }
  }
}
```

Or you can set the value

```
access.property(loginField).set("value", "John Doe")
```

Also you can use shortcut `valueOf` for get values from elements.

2.9. FormData

Above, we considered a method allowing to read values and update form inputs. The flaw of this is the need of reading input one-by-one and lack of files uploading. FormData attended to solve these problems.

In contrast to `property`, FormData works with form tag, not with input tag.

```
val myForm = elementId()
val pictureFieldName = "picture"

form(
  myForm,
  div(
    legend("FormData Example"),
    p(
      label("Picture"),
      input(`type` := "file", name := pictureFieldName)
    ),
    p(button("Submit"))
  ),
  event("submit") { access =>
    access
      .downloadFormData(myForm)
      .flatMap { formData =>
        access.transition { ... }
      }
  }
)
```

See full example

2.10. Delays

Sometimes you want to update a page after a timeout. For example it is useful when you want to show some sort of notification and have to close it after a few seconds. Delays apply transition after a given `FiniteDuration`

```
type MyState = Boolean

def render(state: Boolean) =
  if (state) {
    Html(
      body(
        delay(3.seconds) { access =>
          access.transition(_ => false)
        },
        "Wait 3 seconds!"
      )
    )
  } else {
    Html(
      body(
        button(
          event("click") { access =>
            access.transition(_ => true)
          },
          "Push the button"
        )
      )
    )
  }
}
```

See full example

2.11. Extensions

Korolev is not just request-response, but also push. One of the big advantages of Korolev is the ability to display server side events without additional code. Consider example where we have a page with some sort of notifications list.

```
case class MyState(notifications: List[String])

def render(state: MyState) = optimize {
  ul(
    state.notifications.map(notification =>
      li(notification)
    )
  )
}
```

Also, we have a message queue which has a topic with notifications for current user. The topic has identifier based on `deviceId` of a client. Lets try to bind an application state with messages from the queue.

```
val notificationQueue = ...

val notificationsExtension = Extension { access =>
  for {
    subscription <- notificationQueue subscribe { notification =>
      access.transition(s =>
        s.copy(notifications = notification :: s.notifications)
      )
    }
  } yield Extension.Handlers(onDestroy = subscription.stop())
}

KorolevServiceConfig(
  ...
  extensions = List(notificationsExtension)
)
```

Now every session is subscribed to new notifications. Render will be invoked automatically and user will get his page updated.

See full example

2.12. Routing

Router allows to integrate browser navigation into you app. In the router you define bidirectional mapping between the state and the URL.

Let’s pretends that your application is a blogging platform.

```
case class MyState(author: String, title: String, text: String)

KorolevServiceConfig(
  ...
  // Render the article
  document = {
    case MyState(author, title, text) =>
      Html(
        body(
          div(s"$author: $title"),
          div(text)
        )
      )
  },
  router = Router(
    fromState = {
      case MyState(author, article, _) =>
        Root / author / article
    },
    toState = {
      case Root / author / article => defaultState =>
        articleService.load(author, article).map { text =>
          MyState(author, article, text)
        }
    }
  )
)
```

See full example

2.12.1. Running at a nested path

If Korolev is running at a nested path, e.g. `/ui/`, router's `rootPath` parameter should be set to a desired value:

```
router = Router(
  static = ...,
  dynamic = ...,
  rootPath = "/ui/"
)
```

2.13. Context scope

Large applications have large states with deep hierarchy. For example, when you are making the transition to modify the field of case class which inherits sealed trait, and you know that in this case, you will always have this case class, it can make you fill pain.

```
case class ViewState(tab: Tab)
sealed trait Tab
case class Tab1(...) extends Tab
case class Tab2(...) extends Tab

def renderTab2(tab2: Tab2) = div(
  ...,
  button(
    "Push me",
    event("click") { access =>
      access.maybeTransition {
        // Imagine that you have deeper hierarchy. Pain
        case s @ ViewState(s2: Tab2) =>
          s.copy(tab = s2.copy(...))
      }
    }
  )
)
```

Korolev provides two solutions to solve this problem. First one is Components (noticed above). The second solution is context scoping.

```
val scopedContext = context.scope(
  read = { case ViewState(tab: Tab2) => tab }, // extract sub state
  write = { case (orig, s) => orig.copy(tab = s) } // modify original state
)
```

Now you have context scoped to Tab2. If this context is imported, you will have isolated access to `Tab2`.

Sometimes, being in context scope, you need to apply a transition to all state. In this case, you can handle an event using `eventUnscoped` method of the context. You will get `UnscopedAccess` which ignores current context scope.

See full example

3. JavaScript interoperability

3.1. WebComponents

If you need to extend your application with something that you can't implement with Korolev, you may use Web Components. In simple terms, it is a standard technology that allows you to create custom HTML tags.

NOTE	Korolev doesn't have any specific code to support WebComponents. WebComponent (Custom Elements + Shadow DOM) by design should behave as regular HTML tags. There is no difference between, for example, input or textarea, and any custom element.

See full example

3.2. Call JavaScript from server

Korolev allows executing arbitrary JavaScript code on the client. Use `access.evalJs` to run JavaScript's `eval` method on the client and retrieve the result. The result is a JSON string, so you possibly need to add your favorite JSON library to project dependencies.

```
button(
  "Push me",
  event("click")(_.evalJs("1 + 1").map(println)) // 2
)
```

Also it works for `Promise`.

```
button(
  "Push me",
  event("click") { access =>
    access
      .evalJs(
        """crypto
          |  .subtle
          |  .digest('sha-256', Uint8Array.from([1, 2, 3]))
          |  .then((res) => {
          |    return Array.from(new Uint8Array(res))
          |      .map(x => x.toString(16).padStart(2, '0'))
          |      .join('')
          |  })
        """.stripMargin
      )
      .map(println) //
"039058c6f2c0cb492c533b0a4d14ef77cc0f78abccced5287d84a1a2011cfb81"
  }
)
```

You can refer DOM elements bounded with `ElementId` using `js""` string interpolation.

```
// Declare element
val myElement = elementId()
// Bind the element in template
div(myElement)
// Refer the element
access.evalJs(js"""$myElement.innerHTML = 'this is <a
href="http://example.com">example</a>'""")
```

See full example

3.3. Call server from JavaScript

You can invoke callbacks on a server side from a browser. Declare the callback using `access`.

```
access.registerCallback("myCallback") { myArg =>
  Future(println(myArg))
}
```

Now invoke it from the client side.

```
Korolev.invokeCallback('myCallback', 'myArgValue');
```

See full example

4. Productivity

4.1. Developer mode and "hot reload"

Developer mode provides "hot reload" experience. Run your application with `korolev.dev=true` system property (or environment variable) and session will be kept alive after restart. We recommend to use sbt-revolver plugin.

```
reStart --- -Dkorolev.dev=true
```

Make a change to your app source code and save the file. Switch to the browser and wait for changes to deliver.

Notice that developer mode does not work with custom `StateStorage`.

NOTE	Ensure that everything is stateless except Korolev part of the application. For example, if you keep some state outside of Korolev state, it won't be saved and will lead to inconsistency.

4.2. Serve files from resource directory

Everything placed in directory named `static` (in the classpath of the application) will be served from the `/static/`. It may be useful when you want to distribute some small images or CSS with the app.

WARNING	Korolev is not some sort of CDN node. Avoid serving lots of large files using this feature.

5. Interoperability

5.1. Slf4j

By default Korolev log all messages to standard output. You can redirect logs to SLF4J backend.

Add the dependency.

```
libraryDependencies += "org.fomkin" %% "korolev-slf4j" % "0.16.0"
```

Configure reporter.

```
val config = KorolevServiceConfig(
  ...
  reporter = korolev.slf4j.Slf4jReporter
)
```

5.2. Akka

Korolev provides smooth Akka HTTP integration out of the box. To use it, add a dependency:

```
libraryDependencies += "org.fomkin" %% "korolev-akka" % "0.16.0"
```

```
val service = KorolevServiceConfig[...](
  ...
))
```

And create Korolev route:

```
val config = KorolevServiceConfig[Future, String, Any](
  stateLoader = StateLoader.default("world"),
  document = state => Html(body(div(s"Hello $state")))
)

val korolevRoute = akkaHttpService(config).apply(AkkaHttpServerConfig())
```

Then embed the route into your Akka HTTP routes structure.

See full example

This integration overrides default logging behavior to `akka.event.Logging`.

Despite the fact that Akka uses standard Scala futures, the module can work we any effect type supported by Korolev

5.3. Effects

In addition to standard Scala Futures, Korolev can work with popular effects libraries: ZIO, Cats Effect, and Monix. To use them, add the dependency and set `F` type parameter of` Context` and `KorolevServiceConfig` to demanded effect type.

5.3.1. ZIO

Add dependency for ZIO interop.

```
libraryDependencies += "org.fomkin" %% "korolev-zio" % "0.16.0"
```

Initialize your app with ZIO.

```
import korolev.zio._

implicit val runtime = new DefaultRuntime {}
implicit val zioTaskEffectInstance = taskEffectInstance(runtime)
val ctx = Context[zio.Task, MyState, Any]
val config = KorolevServiceConfig[zio.Task, MyState, Any](...)
```

Korolev uses `Throwable` inside itself. That means if you want to work with your own, you should provide functions to convert `Throwable` to and vice versa.

See full example

5.3.2. Monix

Add Monix interop dependency.

```
libraryDependencies += "org.fomkin" %% "korolev-monix" % "0.16.0"
```

Initialise your app with Monix. Ensure you have implicit `monix.execution.Scheduler` in the scope.

```
import korolev.monix._

val ctx = Context[monix.eval.Task, MyState, Any]
val config = KorolevServiceConfig[monix.eval.Task, MyState, Any](...)
```

See full example

5.3.3. Cats IO

Add Cats interop dependency.

```
libraryDependencies += "org.fomkin" %% "korolev-cats" % "0.16.0"
```

Initialise your app with IO.

```
import korolev.monix._

val ctx = Context[IO, MyState, Any]
val config = KorolevServiceConfig[IO, MyState, Any](...)
```

See full example

6. Troubleshooting

6.1. Debug mode

You view Korolev's client-server exchange. Enter in developer console of your browser and execute this.

```
Korolev.setProtocolDebugEnabled(true)
```

Now you can see something like this.

```
-> [0,0 ]
-> [2,"click",false ]
<- [0,"0:1_3_1_1:click"]
-> [6,"/tab2" ]
-> [4,3,"1_3_1_1",0,"class","checkbox
checkbox__checked",false,0,"1_3_1","1_3_1_2",0,"strike",1,"1_3_1_2","1_3_1_2_1","This
is TODO #0" ]
-> [0,1 ]
```

Message starting with → is commands from server. Message started with ← is callback and client side events. First number is always an procedure or callback code. You can protocol description here.

6.2. Reconnections under Kubernetes Ingress

When using Korolev under Ingress you may face a problem with frequent recconections of websocket channel between browser and server.

For Google Cloud hosting you can try the following:

1. There is section in the Ingress documentaion concerning websockets. It suggests to create a `BackendConfig`.
2. You should set `connectionDraining.drainingTimeoutSec` to sufficiently large value (e.g. `"3600"`), `timeoutSec` doesn't solve the problem.

This solution was tested only for Google Cloud, but it might work for other systems. Don't hesitate to open a PR and describe if this instruction works for other cases.