Korolev: Single Page Applications Framework

Table of Contents

1. Introduction . . . 2
  1.1. Principles . . . 2
  1.2. Quick start . . . 2
2. Understanding Korolev . . . 3
  2.1. Template DSL . . . 3
  2.2. Device and Session . . . 4
  2.3. State . . . 4
  2.4. Render . . . 5
  2.5. Transitions . . . 5
  2.6. Events . . . 6
  2.7. Stateful components . . . 7
  2.8. Access element properties . . . 9
  2.9. FormData . . . 9
  2.10. File streaming . . . 10
  2.11. Delays . . . 11
  2.12. Extensions . . . 12
  2.13. Routing . . . 13
  2.14. Context scope . . . 16
  2.15. Heartbeat and Reconnection . . . 17
  2.16. Emitting Korolev IDs . . . 17
  2.17. Delayed render . . . 17
3. Testing Korolev applications . . . 18
  3.1. Testkit . . . 18
  3.2. Selenium . . . 19
4. JavaScript interoperability . . . 19
  4.1. WebComponents . . . 19
  4.2. Call JavaScript from server . . . 20
  4.3. Call server from JavaScript . . . 21
5. Productivity . . . 21
  5.1. Developer mode and "hot reload" . . . 21
  5.2. Serve files from resource directory . . . 21
6. Interoperability . . . 22
  6.1. Slf4j . . . 22
  6.2. Akka HTTP . . . 22

6.3. Pekko HTTP 23
6.4. Zio-Http 23
6.5. Effects 24
6.6. Streams 25
7. Troubleshooting 27
7.1. Debug mode 28
7.2. Reconnections under Kubernetes Ingress 28

Aleksey Fomkin <a@fomkin.org>

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 11+ installed. Once installed, run g8 template.

```
$ sbt new fomkin/korolev.g8
$ cd mykorolevproject
$ sbt
```

```
> ~reStart
```

The sbt command `~reStart` will make sure that your Korolev application gets restarted and the browser window reloaded on source code changes.

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.

```
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
      }
    )
  }
}
```

As you can observe, the state of the component has a default value. Occasionally, the initial state cannot be set without parameters. In such cases, you can pass a function called `loadState` instead of `initialState`. This function has an effect of `P ⇒ F[S]`, where `P` represents the component parameters. This can be highly advantageous.

Imagine a component that displays the user's display name and photo, while the application state (higher than component) only contains the user's ID. The photo's address and display name are stored in a database.

```
class UserCardComponent(userService: UserService) extends Component[Future, Option
[User], UserId, Any](loadState = userService.getUser(userId)) {

  ...

  // As `User` doesn't load instantly, we should display a placeholder. By default, it
is a simple `div()`.
  def renderNoState(userId: UserId): Node = {
    // Just display user ID while state not loaded
    div(userId.asString)
  }

  // The parameters of the user's ID can be modified, necessitating a state reload
with the updated profile. The function offers an optional effect as a return value. If
no reload is necessary, it returns None; otherwise, it returns the effect with the
updated state.
  def maybeUpdateState(userId: UserId, maybeUser: Option[User]): Option[F[Option[
User]]] =
    maybeUser.collect {
      case user if user.id != userId =>
        userService.getUser(userId)
    }
}

// Inject userService dependency
val UserCard = new UserCardComponent(userService)
```

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
      }
    },
    // And if you don't want to handle event
    MyComponent(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. File streaming

Don’t get confused. Korolev applications works on the server side and client side is remote host. So when we say 'download' it means 'download from client to server'. When we say 'upload' it means 'upload from server to client'.

2.10.1. Downloading

Currently Korolev have few ways to download files from the client side: 1. Download all files, selected within input, as byte arrays. 2. Download all files, but using streams. 3. Manual selection files from input.

In all approaches HTML is the same, you need form with file input and button with handler.

```
Html(
    body(
      input(`type` := "file", multiple, fileInput),
      ul(
        progress.map {
          case (name, (loaded, total)) =>
            li(s"$name: $loaded / $total")
        }
      ),
      button(
        "Upload",
        when(inProgress)(disabled),
        event("click")(onClick)
      )
    )
  )
```

Then you have a methods for each scenario:

1. `downloadFiles` gives you byte arrays.
2. `downloadFilesAsStream` gives you streams.
3. `listFiles` and `downloadFileAsStream` gives you way to select specific file to download.

WARNING `downloadFiles` will store all files in server's RAM until download complete. If you work with large files or huge amount of small files please use streamed approach. Generally streamed approach is a best practice.

```
for {
  files <- access.downloadFilesAsStream(fileInput)
  _ <- Future.sequence {
    files.map { case (handler, data) =>
      val path = Paths.get(handler.fileName)
      data.to(FileIO.write(path))
    }
  }
} yield ()
```

Note that `downloadFilesAsStream` gives you Korolev streams, which is not suitable to be used in applications. Korolev has built-in converters for popular stream libraries; see Streams.

2.10.2. Uploading

You can upload to client side any stream of bytes using `access.uploadFile`.

```
for {
  stream <- effect.Stream("hello", " ", "world").mat()
  bytes = stream.map(s => Bytes.wrap(s.getBytes(StandardCharsets.UTF_8)))
  _ <- access.uploadFile("hello-world.txt", bytes, Some(11L), MimeTypes.`text/plain`)
} yield ()
```

See full example

2.11. 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

2.12. 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

2.13. Routing

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

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

2.13.1. Using Query parameter extractors

Large applications some times require ability to share URL from one user to other with exact some state of sorting and filtering parameters. For that you have possibility to use URL search parameters for state initialisation. For example initialize search form from URI:

```
object BeginOptionQueryParam extends OptionQueryParam("begin")
object EndOptionQueryParam extends OptionQueryParam("end")

case class State(begin: Option[String] = None, end: Option[String] = None)

router = Router(
    fromState = {
      case State(begin, end) =>
        (Root / "search").withParam("begin", begin).withParam("end", end)
    },
    toState = {
      case Root =>
        initialState =>
          Future.successful(initialState)
      case Root  / "search" :?* BeginOptionQueryParam(begin) *&
EndOptionQueryParam(end) => _ =>
          val result = State(begin, end)
          Future.successful(result)
    }
)
```

See full example

For easy parameter matching Korolev provide tree way for parameters matching:

Exact parameter matching, be careful your parameter patterns order and count must have exact same order and count with request:

- `:?` - start query paraters
- `:&` - add one more parameter to query
- `:??` - start exact parameter matchig block

```
val path = Root / "test" :? "k1" -> "v1" :& "k2" -> "v2" :& "k3" -> "v3"
path match {
    case Root / "test" :?? (("k1", v1), ("k2", v2), ("k3", v3)) =>
      (v1, v2, v3)
}
```

Parse parameter manually from Map[String, String]

- `:?*` - extract parameters as Map[String, String]

```scala
val path = Root / "test" :? "k1" -> "v1" :& "k2" -> "v2" :& "k3" -> "v3"
path match {
    case Root / "test" :?* params =>
        ???
}
```

Use parameter extractor syntaxis, combain it with `:?*` for greater results

- `*&` - add one more parameter extractor to match expression

Available parameter extractor:

- `QueryParam` or `QP` - abstract class for matching required request parameter
- `OptionQueryParam` or `OQP`- abstract class for matching optional request parameter

```scala
object K1 extends OQP("k1")
object K2 extends QP("k2")
object K3 extends QP("k3")

val path = Root / "test" :? "k1" -> "v1" :& "k2" -> "v2" :& "k3" -> "v3"
path match {
    case Root / "test" :?* K1(v1) *& K2(v2) *& K3(v3) =>
        ???
}
```

For more information about Path and Querry matching, please read PathAndQuerySpec

2.13.2. 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:

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

2.14. 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.

```scala
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.

```scala
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.

Also you can use similar approach on transition level.

```scala
import korolev.util.Lens

val scopedContext = Lens[ViewState, Tab2](
  read = { case ViewState(tab: Tab2) => tab },
  write = { case (orig, s) => orig.copy(tab = s) }
)

// ...

access.transition(lens) { tab2 =>
  tab2.copy(...)
}
```

See full example

2.15. Heartbeat and Reconnection

By default, the Korolev client-side sends a heartbeat every `5` seconds and tracks how many heartbeats are lost. After two consecutive lost heartbeats, the client performs a forced reconnection to the server. The heartbeat interval can be controlled with `KorolevServiceConfig.heartbeatInterval`. Additionally, you can disable bidirectional heartbeat by setting `KorolevServiceConfig.heartbeatLimit = None`. As a result, the client-side will send messages to the server, but the server will not respond.

2.16. Emitting Korolev IDs

In some cases, you may want to add analytics scripts or other third-party JavaScript to your page that modifies the DOM by adding elements to certain places. Sometimes this can lead to parts of your application ceasing to function. To address this issue, you can set `KorolevServiceConfig.presetIds=true`. As a result, all nodes managed by Korolev will receive an additional attribute `k="6", and Korolev will use these IDs to manage the DOM, ignoring any nodes added via third-party JavaScript.

2.17. Delayed render

In certain scenarios, multiple transitions can occur within a single event handler. If the page is resource-intensive or includes a stateful component with delayed state loading, it can result in decreased performance. However, Korolev has the capability to bundle these transitions together, thereby optimizing rendering.

```
import scala.concurrent.duration.*

val config = KorolevServiceConfig(
  ...
  delayedRender = 2.millis
)
```

The render function will now be called only 2 milliseconds after the initial transition.

NOTE In this case `transitionForce` will wait until delayed render be performed.

3. Testing Korolev applications

3.1. Testkit

Use Korolev Test Kit to simulate environment for event listeners and extensions.

Add Test Kit dependency to your project. You can use use Test Kit with any suitable testing library which supports asynchronous testing.

```
libraryDependencies += "org.fomkin" %% "korolev-testkit" % "1.6.0" % Test
```

Let’s imagine that you are creating application which can to add two integer numbers and to show the result. The app has two text fields and a `span` tag to show the result. The view state of the app is `Option[Int]`. When one of the fields changes, the app reads values of inputs, adds them, and shows calculated result.

```scala
val aInput = elementId()
val bInput = elementId()

def onChange(access: Access) =
  for {
    a <- access.valueOf(aInput)
    b <- access.valueOf(bInput)
    _ <-
      if (a.isBlank || b.isBlank) Future.unit
      else access.transition(_ => Some(a.toInt + b.toInt))
  } yield ()

def renderForm(maybeResult: Option[Int]) =
  form(
    input(
      aInput,
      name := "a-input",
      `type` := "number",
      event("input")(onChange)
    ),
    span("+"),
    input(
      bInput,
      name := "b-input",
      `type` := "number",
      event("input")(onChange)
    ),
    span(s"= ${maybeResult.fold("?")(_.toString)}")
  )
```

Now we can to write test for it. Lets define the environment.

```
val browser = Browser()
  .value(aInput, "2")
  .value(bInput, "3")
```

Check that `onChange` behaves right.

```
"onChange" should "read inputs and put calculation result to the view state" in {
  browser.access(Option.empty[Int], onChange) map { actions =>
    actions shouldEqual List(
      Action.Transition(Some(5))
    )
  }
}
```

Check that event will be handled correctly.

```
it should "be handled" in {
  browser.event(Option.empty[Int],
                renderForm(None),
                "input",
                _.byName("a-input").headOption.map(_.id)) map { actions =>
    actions shouldEqual List(
      Action.Transition(Some(5))
    )
  }
}
```

See full example

3.2. Selenium

Korolev applications like any other web application could be tested using Selenium.

4. JavaScript interoperability

4.1. 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

4.2. 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

4.3. 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

5. Productivity

5.1. 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.

5.2. 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.

6. Interoperability

6.1. 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" % "1.6.0"
```

Configure reporter.

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

6.2. Akka HTTP

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

```
libraryDependencies += "org.fomkin" %% "korolev-akka" % "1.6.0"
```

```
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

6.3. Pekko HTTP

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

```
libraryDependencies += "org.fomkin" %% "korolev-pekko" % "1.6.0"
```

```
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(PekkoHttpServerConfig())
```

Then embed the route into your Akka HTTP routes structure.

See full example

This integration overrides default logging behavior to `org.apache.pekko.event.Logging`.

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

6.4. Zio-Http

Korolev provides smooth Zio-Http integration out of the box.

To use it, add a dependency:

```
libraryDependencies += "org.fomkin" %% "korolev-zio-http" % "1.6.0"
```

And create Korolev service:

```
implicit val effect= new ZioEffect[ZEnv, Throwable](runtime, identity, identity)

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

def route(): HttpApp[ZEnv, Throwable] = {
  new ZioHttpKorolev[ZEnv].service(config)
}
```

Then embed the route into your Zio-Http application.

```
def getAppRoute(): ZIO[ZEnv, Nothing, HttpApp[ZEnv, Throwable]] = {
    ZIO.runtime[ZEnv].map { implicit rts =>
        new Service().route()
    }
}

val prog = for {
  httpApp <- getAppRoute()
  _       <- Server.start(8088, httpApp)
} yield ZExitCode.success
```

See full example

6.5. 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.

6.5.1. ZIO

Add dependency for ZIO interop.

```
libraryDependencies += "org.fomkin" %% "korolev-zio" % "1.6.0"
```

Or ZIO 2

```
libraryDependencies += "org.fomkin" %% "korolev-zio2" % "1.6.0"
```

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

6.5.2. Monix

Add Monix interop dependency.

```
libraryDependencies += "org.fomkin" %% "korolev-monix" % "1.6.0"
```

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

6.5.3. Cats Effect

Add Cats dependency. If you use Cats Effect 2

```
libraryDependencies += "org.fomkin" %% "korolev-ce2" % "1.6.0"
```

Or is you use Cats Effect 3

```
libraryDependencies += "org.fomkin" %% "korolev-ce3" % "1.6.0"
```

Initialise your app with IO.

```
import korolev.cats._

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

See full example

6.6. Streams

Under the hood Korolev uses it’s own simple pull based streams which can work with standard Futures. In the most cases you will not meet them, but sometimes you can. Korolev streams is not suitable to be used in applications, so we offer converters for most popular stream libraries.

6.6.1. Reactive Streams

Reactive Streams is a part of Akka HTTP server integration.

Conversion from Korolev to Publisher.

```
import korolev.akka.instances._

val publisher = myKorolevStream.asPublisher
```

Subscribe to Publisher

```
import korolev.akka.util.KorolevStreamSubscriber

val subscriber = new KorolevStreamSubscriber[F, T]()
publisher.subscribe(result)
```

6.6.2. Akka Streams

Akka streams is a part of Akka HTTP server integration.

Conversion from Akka Stream to Korolev.

```
import korolev.akka.instances._

val akkaSource = korolevStream.asAkkaSource
```

Conversion from Akka Stream to Korolev.

```
import korolev.akka.instances._

val (stream, korolevSink) = Sink.korolevStream[F, String].preMaterialize()

myFlow.to(korolevSink)
```

6.6.3. ZIO

To use it, add a dependency:

```
libraryDependencies += "org.fomkin" %% "korolev-zio-streams" % "1.6.0"
```

Conversion from Korolev to ZIO:

```
val io = KorolevStream(values: _*)
```

```
  .mat[Task]()
  .flatMap { korolevStream: KorolevStream[Task, Int] =>
    korolevStream
      .toZStream
      .run(ZSink.foldLeft(List.empty[Int]){ case (acc, v) => acc :+ v})
  }
```

Conversion from ZIO to Korolev:

```
val values = Vector(1, 2, 3, 4, 5)
val io = ZStream.fromIterable(values)
  .toKorolev()
  .flatMap { korolevStream =>
    korolevStream
      .fold(Vector.empty[Int])((acc, value) => acc :+ value)
      .map(result => result shouldEqual values)
  }
```

6.6.4. Fs2

Conversion from Korolev to Fs2:

To use it, add a dependency.

```
// For Fs2-2.* and Cats Effect 2:
libraryDependencies += "org.fomkin" %% "korolev-fs2-ce2" % "1.6.0"
// For Fs2-3.* and Cats Effect 3:
libraryDependencies += "org.fomkin" %% "korolev-fs2-ce3" % "1.6.0"
```

Conversion from Korolev to Fs2.

```
val io = KorolevStream(values: _*)
  .mat[Task]()
  .toFs2
```

Conversion from Fs2 to Korolev.

```
val values = Vector(1, 2, 3, 4, 5)
val io = fs2.Stream.emits(values)
  .toKorolev
```

7. Troubleshooting

7.1. 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.

7.2. 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.