Udash as the missing link for the type safe web development in Scala

Mateusz Starzec

Development stack - typically

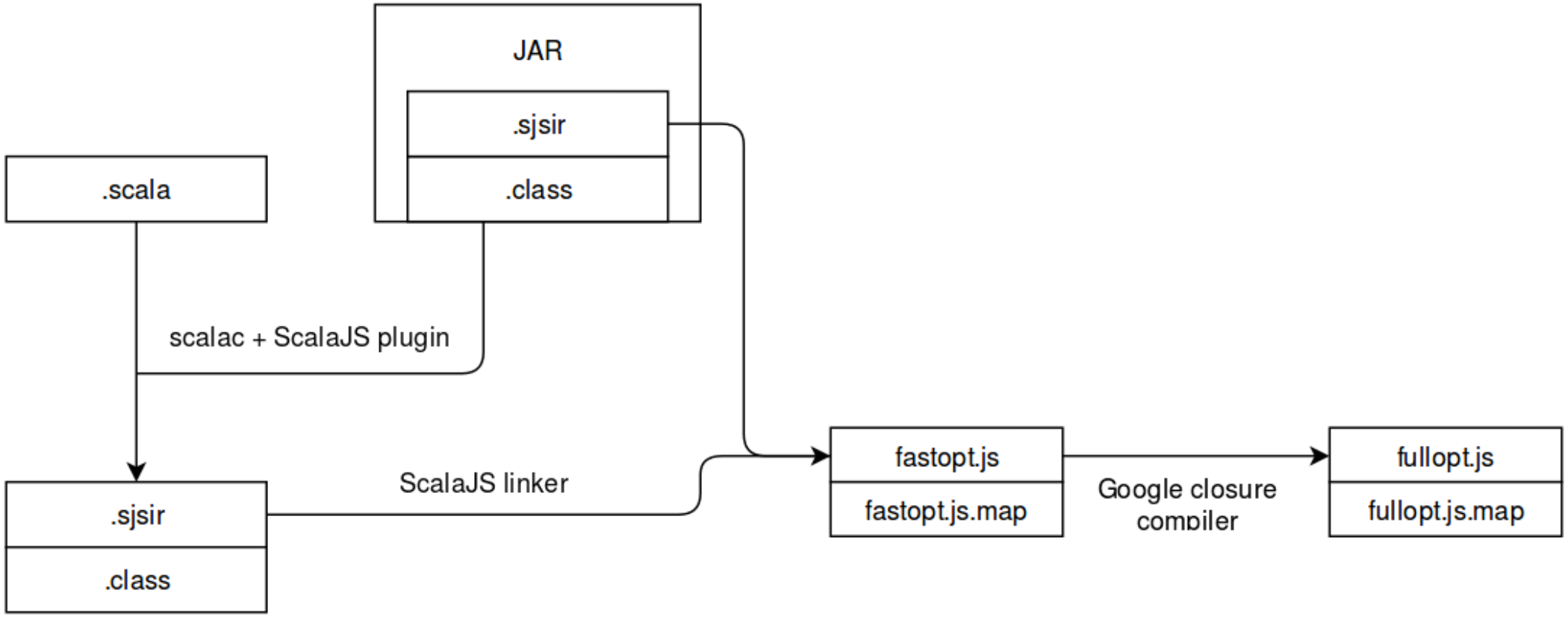
Four different languages:

- Frontend: HTML, CSS, JavaScript
- Backend: Scala

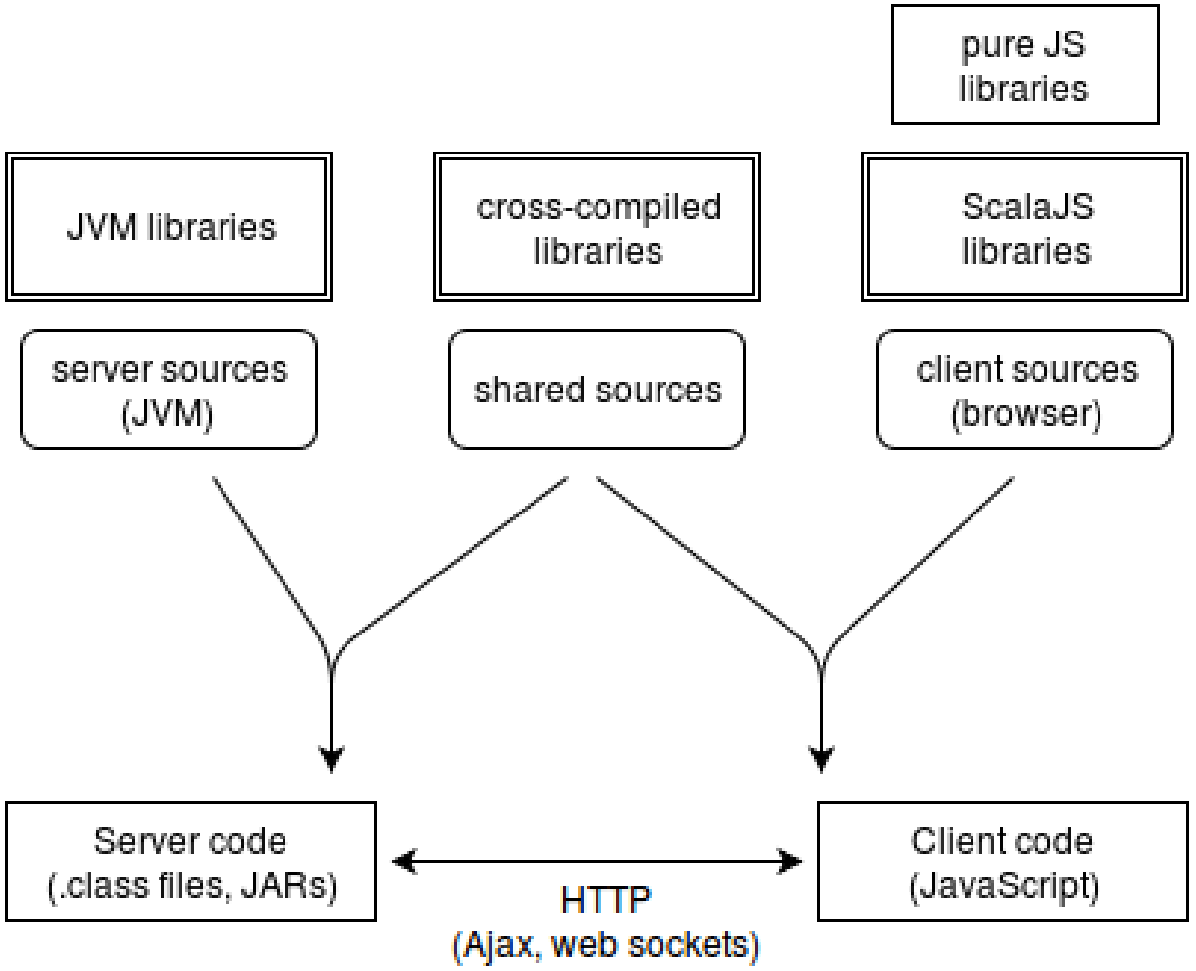
Target:

- Frontend: Scala
- Backend: Scala

scalac+ Scala.js + linker



Project structure



Development stack

We already know how to compile Scala to JavaScript.

What about HTML and CSS?

Scalatags


```
val stack = Seq("Scala", "Scala.js", "Scalatags")
val template = ul(
  stack.map(tool =>
    li(tool)
  )
)
jQ("body").append(template.render)
```

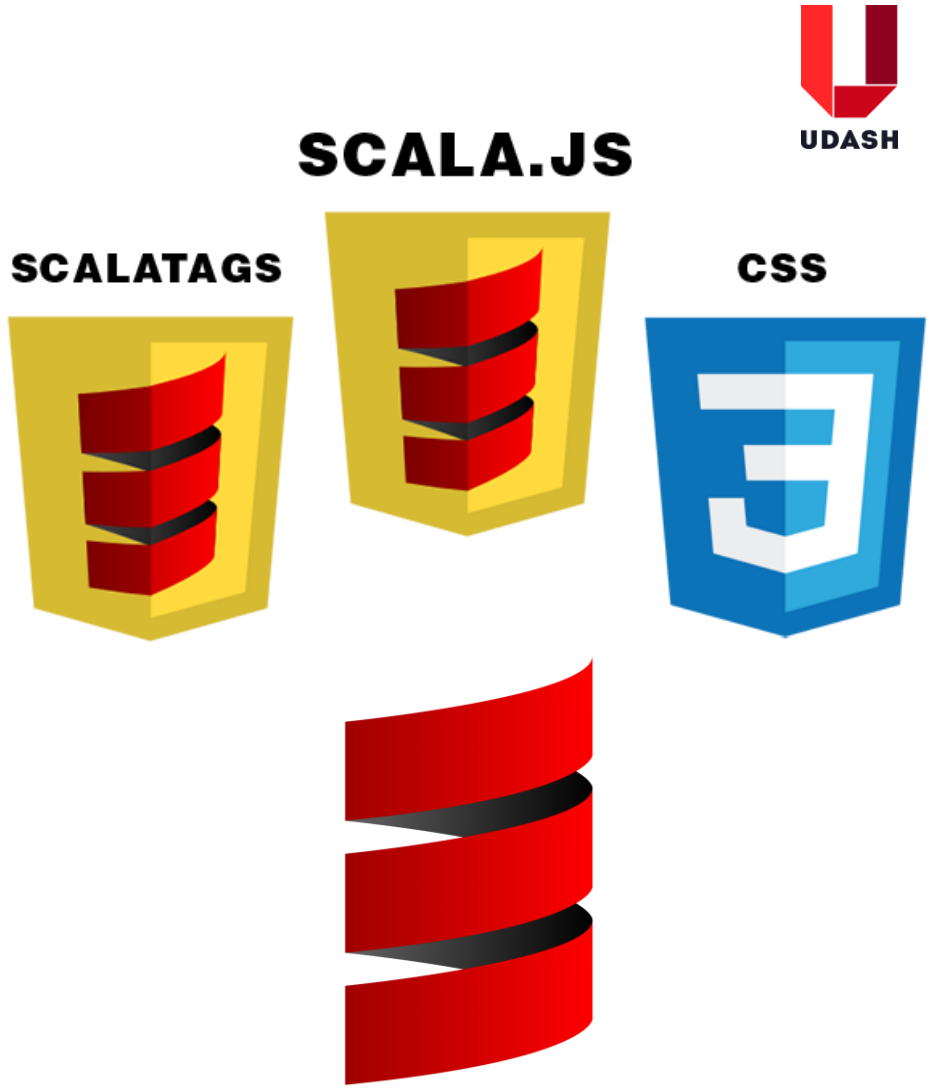
Result:

```
▼<body>
  ▼<ul>
      <li>Scala</li>
      <li>Scala.js</li>
      <li>Scalatags</li>
    </ul>
  </body>
```

Development stack

Scalatags:

- Type-safe API for a dynamic HTML generation in Scala.js
- Powerful templating tool
- It's Scala!



ScalaCSS


```
object ListStyles extends StyleSheet.Inline {
  import dsl._
  val redListItem = style(
    color.red,
    fontSize.large
  )
}

val stack = Seq("Scala", "Scala.js", "Scalatags")
val template = ul(
  stack.map(tool =>
    li(ListStyles.redListItem)(tool)
  )
)
jQ("body").append(ListStyles.render[TypedTag[HTMLStyleElement]].render)
jQ("body").append(template.render)
```

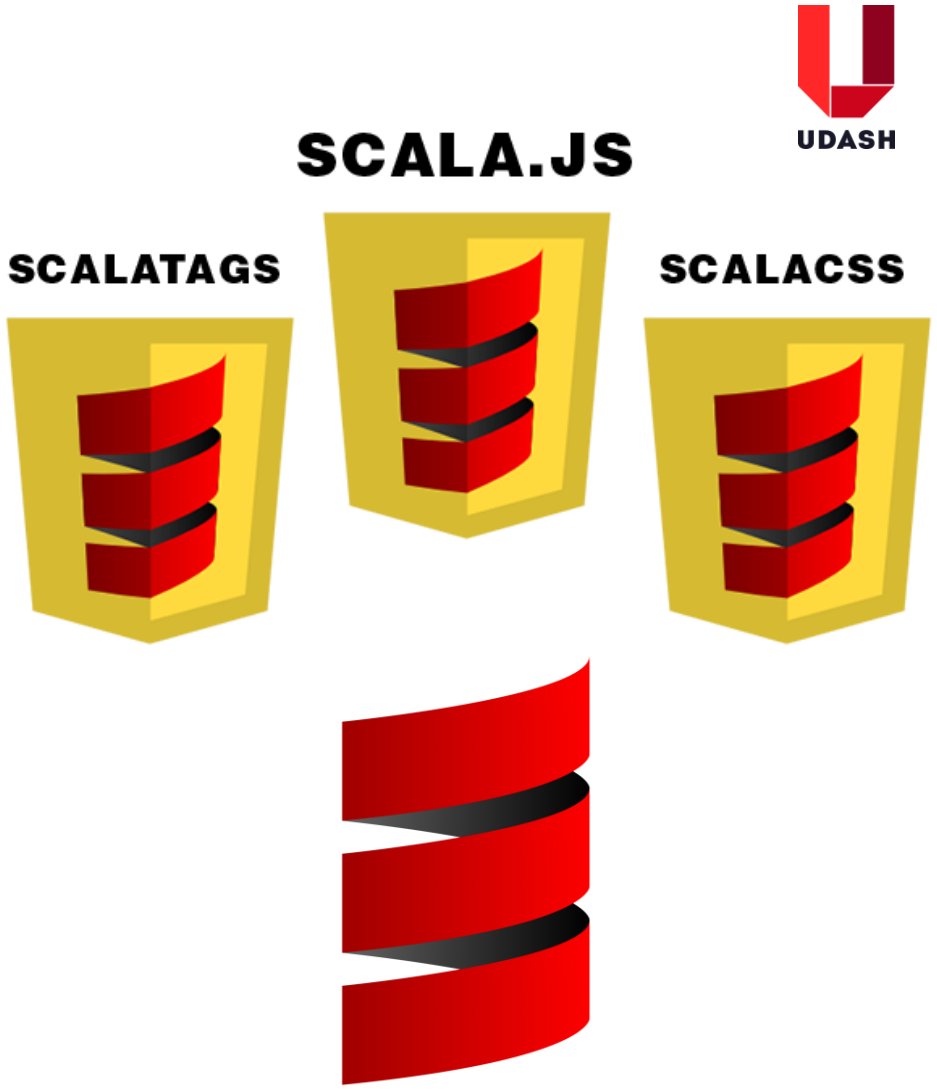
```
▼<body>
  <style type="text/css">.SimpleView_ListStyles_2-redListItem {
    color: red;
    font-size: large;
  }

  </style>
 ▼<ul>
    <li class="SimpleView_ListStyles_2-redListItem">Scala</li>
    <li class="SimpleView_ListStyles_2-redListItem">Scala.js</li>
    <li class="SimpleView_ListStyles_2-redListItem">Scalatags</li>
  </ul>
</body>
```

Development stack

ScalaCSS:

- Type-safe API for a dynamic CSS generation in Scala.js
- Mixins, conflict detection
- Great IDE support

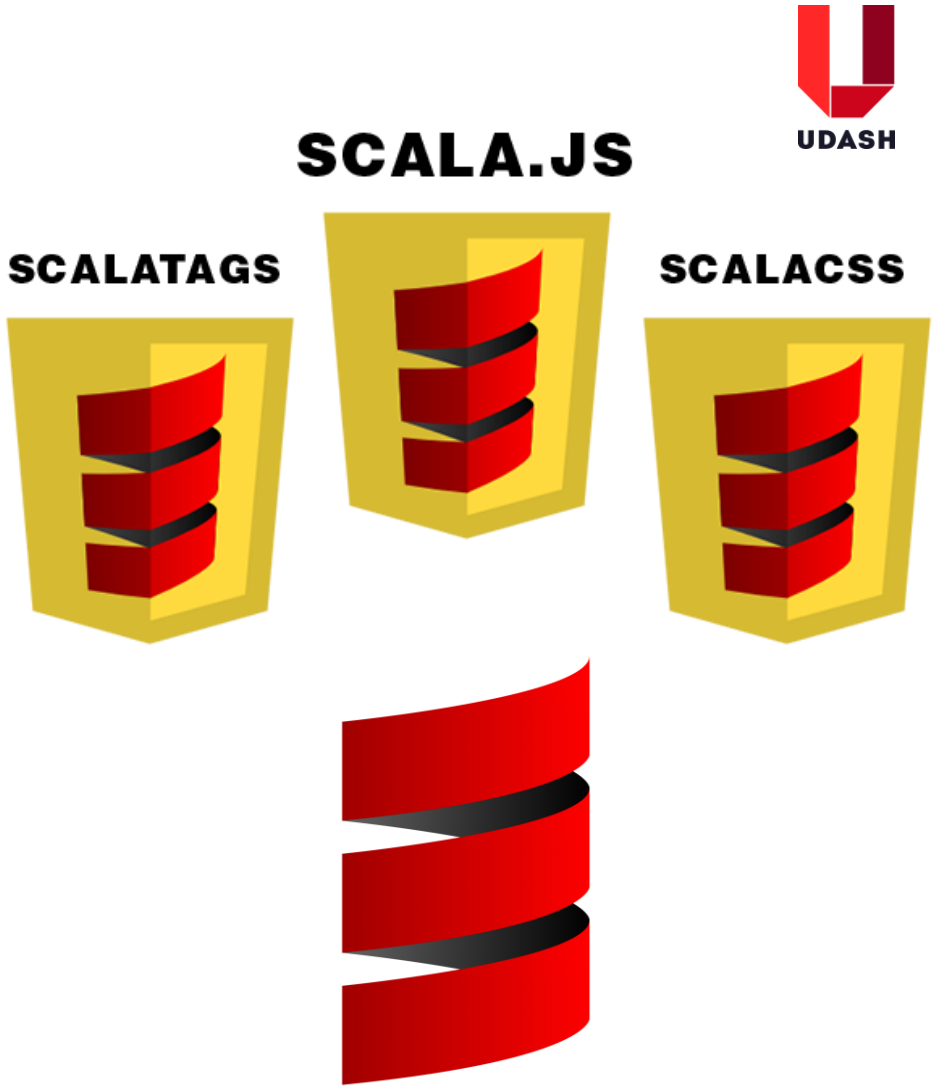


Development stack

Now we have only one language in the backend and the frontend application.

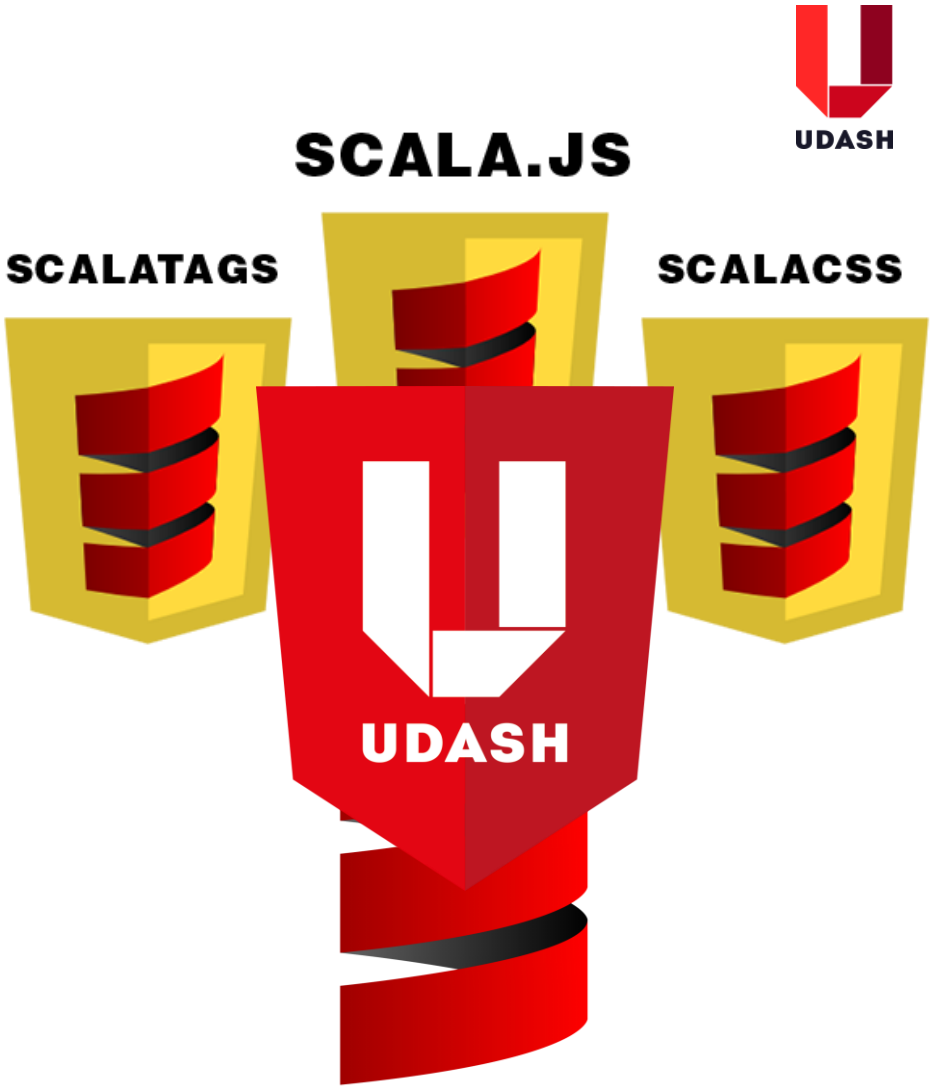
What about:

- data bindings?
- frontend routing?
- client ↔ server communication?
- i18n?



Udash Framework

- The Udash modules:
  - Udash Core
  - Udash RPC
  - Udash i18n



Udash Core - Properties

The Udash data model system:

- Property – contains single, immutable value
- ModelProperty – complex structure containing other properties
- SeqProperty – sequence of the properties

Features:

- Value change listeners
- Asynchronous validation
- Transformation and SeqProperty filtering

Udash Core - Properties


```scala
trait NumbersInRange {
  def minimum: Int
  def maximum: Int
  def numbers: Seq[Int]
}

val numbers = ModelProperty[NumbersInRange]
numbers.subProp(_.minimum).set(0)
numbers.subProp(_.maximum).set(42)

val s: SeqProperty[Int] = numbers.subSeq(_.numbers)
s.set(Seq(3,7,20,32))
// s.get == Seq(3, 7, 20, 32)
s.replace(idx = 1, amount = 2, values = 8,9,10)
// s.get == Seq(3, 8, 9, 10, 32)
```

Udash Core - Properties


```
object UserNameValidator extends Validator[String] {
  def apply(name: String)
           (implicit ec: ExecutionContext): Future[ValidationResult] =
    Future {
      if (name.length >= 3) Valid
      else Invalid(Seq("User name must contain at least 3 characters!"))
    }
}

val name = Property[String]
name.addValidator(UserNameValidator)
name.set("A")
name.isValid // returns Future(Invalid)
name.set("Abcde")
name.isValid // returns Future(Valid)
```

Udash Core - Properties

Properties Demo

Udash Core - Properties


```
val doubles = SeqProperty[Double](Seq(1.5, 2.3, 3.7))
val ints = doubles.transform((d: Double) => d.toInt, (i: Int) => i.toDouble)
val evens = ints.filter(_ % 2 == 0)
doubles.listen((s: Seq[Double]) => println(s"Doubles: $s"))
ints.listen((s: Seq[Int]) => println(s"Ints: $s"))
evens.listen((s: Seq[Int]) => println(s"Evens: $s"))
println("---")
doubles.append(6.5)
println("---")
ints.prepend(-3)
```

Prints out:

```
---
Evens: ListBuffer(2, 6)
Doubles: ListBuffer(1.5, 2.3, 3.7, 6.5)
Ints: ListBuffer(1, 2, 3, 6)
---
Doubles: ListBuffer(-3, 1.5, 2.3, 3.7, 6.5)
Ints: ListBuffer(-3, 1, 2, 3, 6)
```

Udash Core - Properties

Advanced Properties Demo

Udash Core - Bindings

Binding Properties into the Scalatags templates:

- Automatically updates presented element after property change
- Binds property value into DOM element attribute
- Binds property validation result
- Binds user input from form elements into properties

Udash Core - Bindings


```
val name = Property("World")
val element= div(
  TextInput(name), br,
  produce(name)(userName => h3(s"Hello, $userName!").render)
).render
jQ("body").append(element)
```

Result:

Udash

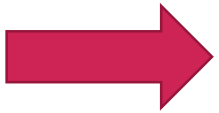
Hello, Udash!

Udash Core - Bindings


```
val doubles = SeqProperty[Double](Seq(1.5, 2.3, 3.7))
val ints = doubles.transform((d: Double) => d.toInt, (i: Int) => i.toDouble)
val evens = ints.filter(_ % 2 == 0)
val element = ul(
  li("Doubles: ", repeat(doubles)((el: Property[Double]) => span(s"${el.get} ").render)),
  li("Ints: ", repeat(ints)((el: Property[Int]) => span(s"${el.get} ").render)),
  li("Evens: ", repeat(evens)((el: Property[Int]) => span(s"${el.get} ").render))
).render
jQ("body").append(element)
doubles.insert(idx = 1, values = 8.5)
```

Result:

- Doubles: 1.5 2.3 3.7
- Ints: 1 2 3
- Evens: 2



- Doubles: 1.5 8.5 2.3 3.7
- Ints: 1 8 2 3
- Evens: 8 2

Udash Core - Bindings

Binding Demo

Udash Core - Application

- Starting point of the frontend code
- Application::run(attachToElement: dom.Element)
  - Resolves the initial application state and view
  - Adds resolved view to the DOM hierarchy as a child of the provided node
- The root node has to be present in the bootstrapping HTML

Udash Core - Application

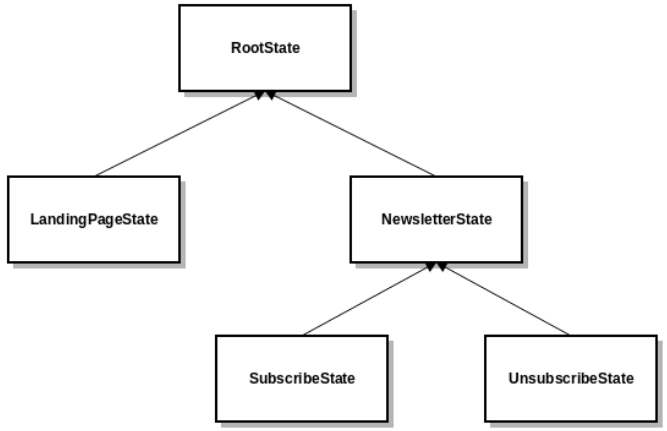

```
val applicationInstance = new Application[RoutingState](routingRegistry, viewPresenterRegistry, RootState)
jQ(document).ready((_: Element) => {
  val appRoot = jQ("#application").get(0)
  if (appRoot.isEmpty) {
    logger.error("Application root element not found! Check your index.html file!")
  } else {
    applicationInstance.run(appRoot.get)
  }
})
```

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>agh-demo</title>

    <script src="scripts/frontend-deps.js"></script>
    <script src="scripts/frontend-impl.js"></script>
    <script src="scripts/frontend-init.js"></script>
</head>
<body>
  <div id="application"></div>
</body>
</html>
```

Udash Core - State

- Type-safe equivalent of the URL
- Hierarchical structure
- Views resolving is based on the states




```
sealed abstract class RoutingState(val parentState: RoutingState) extends State
case object RootState extends RoutingState(null)
case object LandingPageState extends RoutingState(RootState)
case object NewsletterState extends RoutingState(RootState)
case object SubscribeState extends RoutingState(NewsletterState)
case object UnsubscribeState extends RoutingState(NewsletterState)
```

Udash Core - Routing

- Routing based on the URL part following # sign
- Resolves an application state basing on the URL
- Resolves a view basing on the state

```
val (url2State, state2Url) = Bidirectional[String, RoutingState] {
  case "/users" => Dashboard
  case "/users/search" => UsersListState(None)
  case "/users/search" /:/ query => UsersListState(Some(query))
  case "/users/details" /:/ username => UserDetailsState(username)
}
```

Udash Core – Application & Routing

Application & Routing Demo

Udash Core - Views

- Typical view has:
  - Model structure definition

```
trait NumbersInRange {
  def minimum: Int
  def maximum: Int
  def numbers: Seq[Int]
}

val model = ModelProperty[NumbersInRange]
```

Udash Core - Views

- Typical view has:
  - Model structure definition
  - *Presenter* – business logic

```
class PropertiesDemoPresenter(model: ModelProperty[NumbersInRange]) extends Presenter[PropertiesDemoState.type] {
  private val minimum: Property[Int] = model.subProp(_.minimum)
  private val maximum: Property[Int] = model.subProp(_.maximum)
  private val numbers: SeqProperty[Int] = model.subSeq(_.numbers)

  override def handleState(state: PropertiesDemoState.type): Unit = ()

  def randomize(): Unit = {
    minimum.set(Random.nextInt(20))
    maximum.set(minimum.get + Random.nextInt(80))
    numbers.set(Seq.fill(3)(Random.nextInt(100)))
  }
}
```

Udash Core - Views

- Typical view has:
  - Model structure definition
  - *Presenter* – business logic
  - *View* – Scalatags template, child view rendering

```
class PropertiesDemoView(model: ModelProperty[NumbersInRange],
                         presenter: PropertiesDemoPresenter) extends View {
  override def getTemplate: Element = ???
  override def renderChild(view: View): Unit = ???
}
```

Udash Core - Views

- Typical view has:
  - Model structure definition
  - *Presenter* – business logic
  - *View* – Scalatags template, child view rendering
  - *ViewPresenter* – factory of the model, the view and the presenter

```
object PropertiesDemoViewPresenter extends ViewPresenter[PropertiesDemoState.type] {
  override def create(): (View, Presenter[PropertiesDemoState.type]) = {
    val model = ModelProperty[NumbersInRange]
    val presenter = new PropertiesDemoPresenter(model)
    val view = new PropertiesDemoView(model, presenter)
    (view, presenter)
  }
}
```

Udash Core – Views

Views Demo

Udash Core

Summary:

- Strongly typed web framework
- Asynchronous design
- Written in Scala.js
- Open source

Udash RPC

Client ↔ server communication system:

- Works in the both directions out of the box
- Asynchronous by default
- Crosscompiled traits describing the RPC interface
- Strongly typed interfaces, checked in compile time
- Usage looks like standard method call
- Based on the websocket

Udash RPC - Interfaces

Methods exposed by the RPC interface can be:

- ***Calls*** - methods returning *Future[T]* where *T* is a serializable type (a client RPC interface cannot expose those methods)
- ***Fires*** - methods with a return type *Unit* there is no guarantee that your request will be received by a recipient
- ***Getters*** - methods returning another RPC interface, calling this method does not send anything over the network

Udash RPC - Interfaces

Shared:

```scala
@RPC
trait ServerRPC {
  def fire(): Unit
  def call(yes: Boolean): Future[String]
  def innerRpc(name: String): InnerRPC
}

@RPC
trait InnerRPC {
  def innerCall(arg: Int): Future[String]
}
```

Udash RPC – client calls server

Server-side implementation:

```
class ExposedRpcInterfaces(implicit clientId2: ClientId) extends ServerRPC {
  def fire(): Unit = println("fire")
  def call(yes: Boolean): Future[String] = Future {
    if (yes) "Yes" else "No"
  }
  def innerRpc(name: String): InnerRPC = new DummyInnerRPC(name)
}

class DummyInnerRPC(name: String) extends InnerRPC {
  def innerCall(arg: Int): Future[String] = Future { name * arg }
}
```

Client-side call:

```
val serverRpc = DefaultServerRPC[MainClientRPC, ServerRPC](new RPCService)
serverRpc.innerRpc("Udash").innerCall(3) onComplete {
  case Success(response) => println(response)
  case Failure(ex) => ex.printStackTrace()
}
```

Udash RPC – server calls client

Client-side implementation:

```
class FrontendRPCService extends ClientRPC {
  override def clientFire(i: Int): Unit =
    println(s"Server called clientFire($i)")
}
```

Server-side call:

```
val clientId: ClientId = ???
ClientRPC(AllClients).clientFire(10)
ClientRPC(clientId).clientFire(20)
```

Udash RPC

Server↔Client RPC Demo

Udash i18n

- Typed translation keys arguments
- Two ways of translations providing:
  - Compiled into frontend JavaScript code
  - Served by backend application via RPC (with frontend caching)
- Property-based translation binding – dynamic language change

Udash i18n


```
object Translations {
  import TranslationKey._

  object auth {
    val loginLabel = key("auth.loginLabel")
    val passwordLabel = key("auth.passwordLabel")

    object login {
      val buttonLabel = key("auth.login.buttonLabel")
      val retriesLeft = key1[Int]("auth.login.retriesLeft")
    }

    object register {
      val buttonLabel = key("auth.register.buttonLabel")
    }
  }
}
```

Templates:

```
auth.login.buttonLabel=Sign in
auth.login.retriesLeft={} retries left
```

Udash i18n

Usage:

```
object FrontendTranslationsProvider {
  private val translations = Map(
    Lang("en") -> Bundle(BundleHash(""), Map(
      "auth.login.buttonLabel" -> "Sign in",
      "auth.login.retriesLeft" -> "{} retries left"
    ))
  )

  def apply(): LocalTranslationProvider =
    new LocalTranslationProvider(translations)
}

implicit val translationProvider = FrontendTranslationsProvider()
implicit val lang = Lang("en")
div(
  ul(
    li("auth.login.buttonLabel: ", translated(Translations.auth.login.buttonLabel())),
    li("auth.login.retriesLeft: ", translated(Translations.auth.login.retriesLeft(3))),
    li("auth.login.retriesLeft: ", translated(Translations.auth.login.retriesLeft("three")))
  )
).render
```

Udash i18n

Translation Demo

Udash Generator

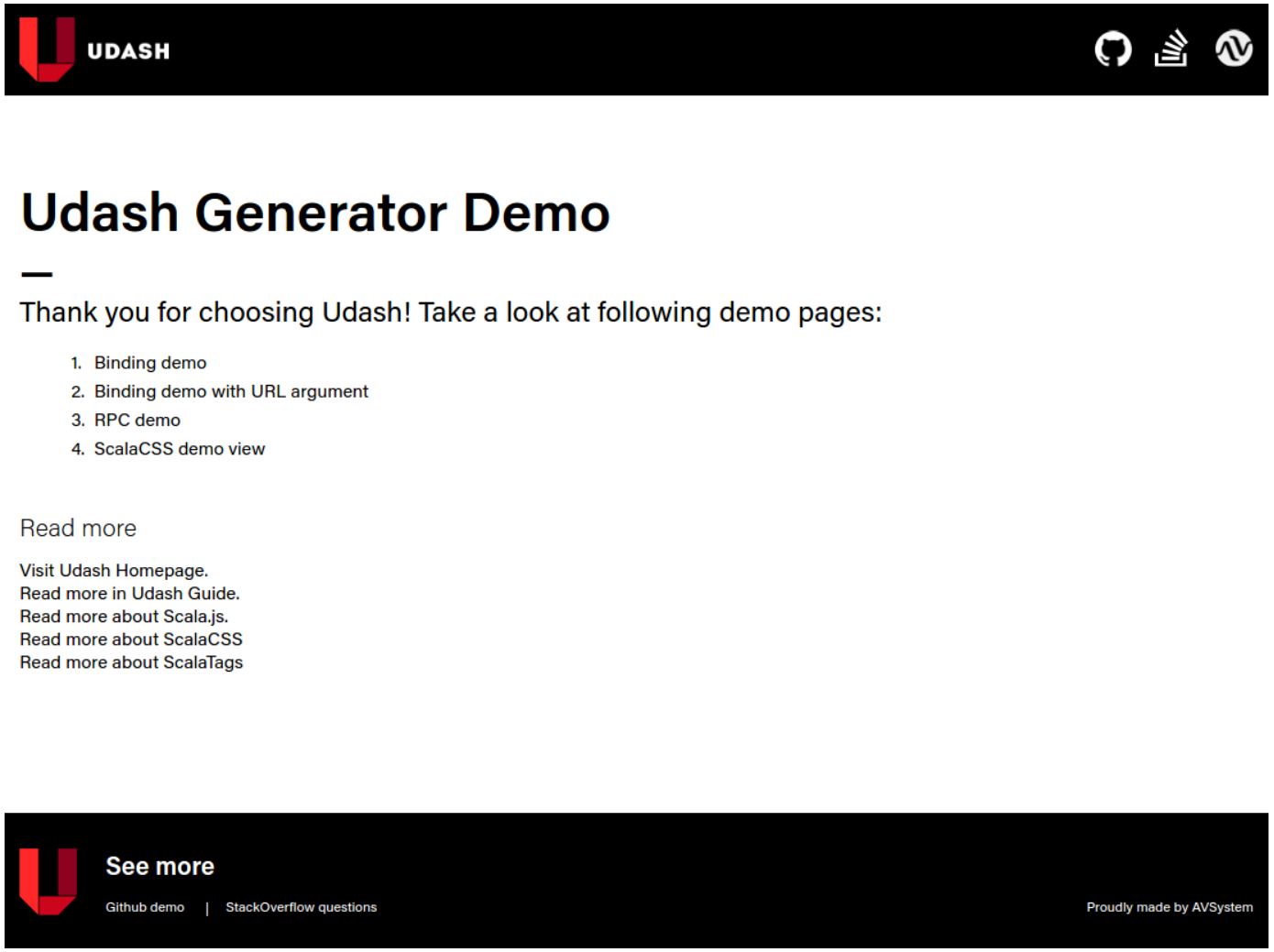
Quickly bootstrap Udash based project:

- Generator creates full, ready to compile project
- You can select which libraries you want to use
  - Udash
  - Udash RPC
  - ScalaCSS
  - Jetty

Udash Generator

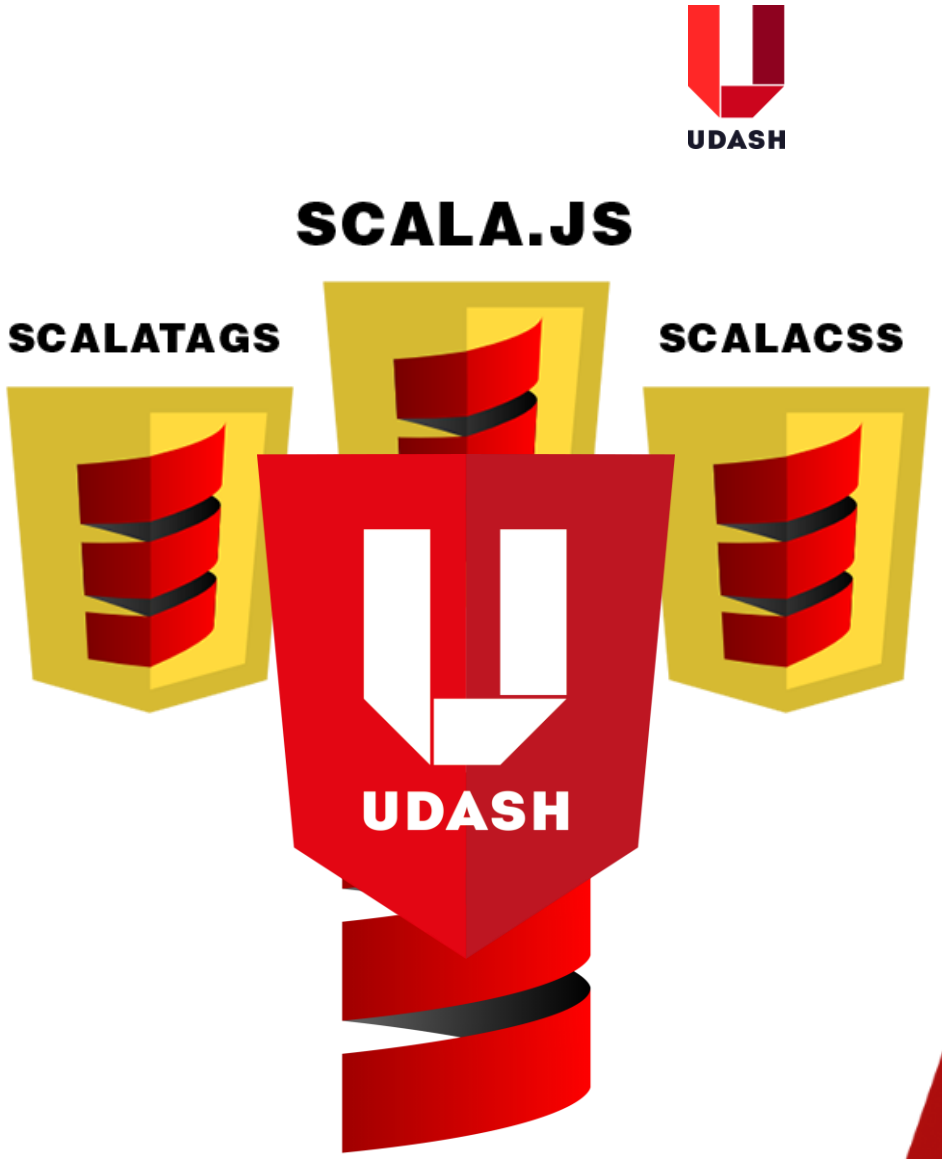

```
Project root directory [/home/starzu/uidas/generator/dist/udash-app]: /home/starzu/udash-demo
Clear root directory [false]: true
Project name [udash-app]: udash-demo
Organization [com.example]: io.udash
Root package [com.example]: io.udash.demo
Project type [1]:
  1. FrontendOnlyProject
  2. StandardProject(backend,shared,frontend)
Select: 2
Backend module name [backend]:
Shared module name [shared]:
Frontend module name [frontend]:
Create basic frontend application [true]:
Create frontend demo views [true]:
Create ScalaCSS demo views [true]:
Create Jetty launcher [true]:
Create RPC communication layer [true]:
Create RPC communication layer demos [true]:
Start generation [true]:
```

Udash Generator



Summary

- Scala everywhere + shared code
- Types wherever possible
- Quick bootstrapping
- Easy development



Thank you