Leon Framework Documentation

Release 0.2

WeigleWilczek GmbH

November 19, 2011

CONTENTS

1 What is Leon? 3

2 Introduction to Leon 5
2.1 Getting started . . . 5

3 Leon Core 7
3.1 Logging . . . 7
3.2 Configuration . . . 7
3.3 Java Interoperability . . . 8

4 Browser 11
4.1 Call server-side functions (AJAX) . . . 11
4.2 Receive messages from server (Comet) . . . 11

5 Modules 13
5.1 Closure Templates . . . 13
5.2 CoffeeScript . . . 13
5.3 MongoDB . . . 13
5.4 SQL . . . 14

6 IDE Support 15
6.1 Leon Eclipse Plugin . . . 15

Index 17

Version 0.2

Contents:

CHAPTER

ONE

WHAT IS LEON?

Leon is an application framework for building smart web applications. We believe that in modern web applications the UI should be completely based on HTML(5), CSS and JavaScript, and be rendered in the browser while the server should provide strong AJAX / Comet communication mechanisms, database access, security management, etc.

Leon combines and integrates best-of-breed client frameworks like jQuery and AngularJS with it's strong server runtime. Even though Leon is JVM-based (implemented in Scala & Java), server-side code can be written in JavaScript as well. Leon's Java/JavaScript integration goes beyond simple method calls by providing automatic conversions of JavaScript objects and Java POJOs, for example.

For those who like it (we certainly do!), we provide automatic CoffeeScript and Less CSS generation. For example, CoffeeScript files can be used as a drop-in replacement and be used on the server- as well as client-side.

We decided to use the JVM as the development and deployment target instead of e.g. node.js so that we can easily connect to databases, message brokers and even SAP systems. The JVM is probably the most proven and tested application platform.

Why a chameleon as as mascot, you ask? Well, applications written with Leon may utilise Java, Scala, JavaScript, CoffeeScript and Less CSS, programmers might start with a Java JAR file providing the application logic or use Leon's HTML form driven approach for rapid CRUD applications. Hence, working with Leon can look quite differently while it's all the same in the end.

CHAPTER

TWO

INTRODUCTION TO LEON

TODO

2.1 Getting started

TODO

CHAPTER

THREE

LEON CORE

3.1 Logging

Leon provides a small wrapper around SLF4J (TODO: insert URL) to ease the usage for JavaScript code.

`leon.`**`getLogger`**(*name*)
Creates a slf4j logger instance with the given name.

Arguments

- **name** (*String*) – The name of the logger.

3.2 Configuration

TODO

`install`(*module*)
Installs additional Leon modules.

Arguments

- **module** (*Module*) – The module to install. (e.g. `new Packages.io.leon.persistence.mongo.LeonMongoModule()`)

`setBaseDir`(*baseDir*)
Sets the base directory of the application. All relative locations are relative to this directory.

Arguments

- **baseDir** (*string*) – Default value is the location of the configuration file.

3.2.1 Resources

`addLocation`(*path*)
Adds a directory where Leon should lookup resources.

Arguments

- **path** (*string*) – Relative paths are relative to the base directory.

`exposeUrl`(*regex*)
Exposes paths that match the given regular expression to clients.

By default, Leon gives client access to the following paths/files: `*.html`, `*.png`, `*.jpg`, `*.gif`, `*.css`, `favicon.ico`, `*/browser/*.js`, `*/browser/*.json`. To allow access on other paths or files to clients, you have to expose them by calling this function.

Arguments

- **regex** (*string*) – Regular expression to match.

`loadFile` (*fileName*)
Loads JavaScript files in the server environment.

Arguments

- **fileName** (*string*) – A server-side JavaScript file to load. The path must be relative to a registered location or to the application's classpath.

3.2.2 AJAX-Support

`browser` (*browserName).linksToServer([serverName]*)
Makes a server-side object accessible by clients via AJAX.

Arguments

- **browserName** (*string*) – Name of the client-side variable.
- **serverName** (*string*) – Variable name of the server-side object. If not given, serverName is the same as browser name.

`browser` (*browserName).linksToServer(clazz*)
Makes Java objects directly accessible by clients via AJAX.

Arguments

- **browserName** (*string*) – Name of the client-side variable.
- **clazz** (*Class*) – Java class on the server-side.

3.2.3 Dependency injection

`bind` (*clazz*)
TODO: Registers a binding in google guice. See Google Guice Binder for more information about how to use bindings.

Arguments

- **clazz** (*Class*) – The Java class to bind.

3.3 Java Interoperability

Leon uses Mozilla's Rhino [1] JavaScript engine which comes with java interoperability out-of-the-box. This enables us to [2]:

- create instances of Java classes
- call methods of Java objects
- access bean properties as they were ordinary attributes

[1] Mozilla Rhino http://www.mozilla.org/rhino/
[2] Scripting Java http://www.mozilla.org/rhino/scriptjava.html

- extend java classes and implementing interfaces in JavaScript.

However, this interoperability is limited to primitive types only. Leon enables us to call methods of Java objects with complex JSON data structures by transforming them to the corresponding java type. For Java, this feature is limited to POJOs [3] or simple Java Beans and Java collection types.

3.3.1 Using Java objects in JavaScript

You can create a new instance of a Java object by using the keyword `new` and the full-qualified class name prefixed with `Packages`:

```
var obj = new Packages.java.lang.StringBuffer("I'm a Java object");
```

Another way to access a Java object in JavaScript is by asking the dependency injector to get a reference to an object:

```
var obj = leon.inject(Packages.xyz.MyJavaObject);
```

Getters and Setters can be accessed as they were ordinary attributes. Instead of `person.getName()` you can write `person.name` and instead of `person.setName(x)` you can write `person.name = x` in JavaScript.

JavaScript objects can be passed as arguments to Java methods and will be converted to the corresponding Java type automatically. However, this will not work for overloaded method calls. It would not be guaranteed that Leon selects the desired method. In that case you have to perform the serialization manually by calling the method `asJavaObject(clazz)` yourself.

```
var obj = {...}.asJavaObject(Packages.io.leon.test.TestBean);
javaObject.overloadedMethod(obj, 123);
```

Note that return values will not be converted automatically, because we don't want to destroy its identity. They are wrapped in a transparent proxy type and you can work with it like it were a ordinary JavaScript object. If you pass such a java proxy to a method, the java object gets simply unwrapped and there is no need for a complete conversion. However, to convert such a proxy object to a native JavaScript object, you can call the `toJSON` method.

3.3.2 Serializing Java objects to JSON

Every Java object you work with in JavaScript has a function called `toJSON`. It returns a JSON representation of the object by converting each property to the corresponding JavaScript type. This is most suitable for data objects more specifically for POJOs.

If a java object is part of an HTTP response this function is called automatically. So in most cases you can ignore this function, but it's good to know how to convert a Java object to a native JavaScript object anyway.

The following conversions are supported:

- Primitive java types (including the corresponding `java.lang` objects) to JavaScript
- Java arrays to JavaScript arrays
- Java collection types to JavaScript arrays (Currently all `java.util.Collection` types are supported but not `java.util.Map`)
- POJOs to JavaScript objects

Note: Dates are currently not supported.

[3] POJO (Plain Old Java Object) http://en.wikipedia.org/wiki/POJO

3.3.3 Serializing JSON to Java objects

The serialization of JavaScript to Java is triggered automatically when a method of a Java object is called from JavaScript. More precisely, the provided arguments are converted. For instance, you can call a Java method which expects some kind of POJO as its argument from JavaScript. Leon tries to build that object from the supplied JavaScript object.

The serialization can also be triggered manually by calling the method `asJavaObject(clazz)` where `clazz` is the desired target type. This method is available on all JavaScript objects.

The following conversions are supported:

- primitive JavaScript types to Java (Note: Date is currently not supported)
- JavaScript objects to POJOs
- JavaScript arrays to Java collections (`java.util.Map` is currently not supported)

  Leon looks up the desired Java collection type, tests if it is assignable from `java.util.List` or `java.util.Set` and converts the JavaScript array to an implementation of that type.

  Additionally, the following concrete Java Collection types are supported which can be used in signatures:

  - `java.util.ArrayList`
  - `java.util.LinkedList`
  - `java.util.Vector`
  - `java.util.HashSet`
  - `java.util.TreeSet`
  - `java.util.LinkedHashSet`

  For unsupported collection types Leon raises an exception.

CHAPTER

FOUR

BROWSER

4.1 Call server-side functions (AJAX)

TODO

4.2 Receive messages from server (Comet)

It's a common requirement to modern Web applications to retrieve messages without the browser explicitly requesting it. A wide known umbrella term for that is Comet [1]. Leon makes it very easy to make use of that technique.

A client can subscribe to one or more topics and registers a handler function which gets called for each new message. You can also set and update filters to only receive messages which apply to a specific filter rule.

4.2.1 Subscribe to a topic to retrieve messages

To subscribe to a topic in your html page, Leon offers a tag called `<leon:subscribe/>` which can be placed inside your code. For every page request, Leon registers that unique page to the topic(s) and replaces all `<leon:subscribe/>` tags with the necessary JavaScript code. A simple html page with a subscription would look like that:

```
<html xmlns:leon="http://leon.io">
  <head>
    <title>Test Page</title>
    <!-- <#include "/io/leon/templates/full.desktop.html" /> -->
    <script type="text/javascript">
      // <![CDATA[
      function myCallback(message) {
        alert(message);
      }
      // ]]>
    </script>
  </head>
  <body>
    <leon:subscribe topic="myChannel" handlerFn="myCallback" />
  </body>
</html>
```

[1] Comet http://en.wikipedia.org/wiki/Comet_%28programming%29

In this example the page subscribes to a topic called `myChannel` and it registers a callback function called `myCallback`. The function gets called every time the page receives a new message from the server. In this example, an alert box will be prompted to the user.

If you publish a message to a topic, every user who stays on a page which is subscribed to that topic will receive the message. Sometimes this is not what you want! If so, feel free to read on to learn something about message filtering.

4.2.2 Filter messages

Let's say you are only interested in specific messages in a topic based on settings the user has made in the UI. For that case, you can set and update some filter rules. All filters will be applied on the server-side and only messages which have passed the filter will be send to the client.

Filter values are valid for one unique page view. Means, that every single requested page has its own filter rules.

To set a filter, you have to declare the fields you are interested in with the attribute `filterOn` in `<leon:subscribe />` first. Here you specify the field names seperated by comma.

```
<leon:subscribe topic="myChannel" filterOn="field1, field2" handlerFn="myCallback" />
```

To set or update the actual filter values, you call the `leon.comet.updateFilter` JavaScript function.

```
leon.comet.updateFilter("myChannel", "field1", "1")
leon.comet.updateFilter("myChannel", "field2", "2")
```

Note: You can only update filters on fields which are declared in the `filterOn` attribute.

CHAPTER

FIVE

MODULES

5.1 Closure Templates

Leon uses Googles's [1] Closures Templates to enable client-side templating.

5.1.1 Defining a client-side template

Closures Templates are defined in `.soy` files. See the official documentation [2] for further information about the `.soy` syntax. Template files have to be placed in a *Resources* folder.

5.1.2 Automatic compilation of `.soy` files

All `.soy` files are automatically compiled by the Closures compiler. The result is a `.js` file with the same name at the same location as the `.soy` file.

5.2 CoffeeScript

TODO

5.3 MongoDB

This Leon module enables your application to interact with a MongoDB [3].

5.3.1 Enable Leon's MongoDB module

Leon's MongoDB module can be enabled by adding the following line to Leon's configuration file:

```
install(new Packages.io.leon.persistence.mongo.LeonMongoModule());
```

[1] Closure Templates http://code.google.com/intl/de/closure/templates/
[2] Closure File Structure http://code.google.com/intl/de/closure/templates/docs/concepts.html#filestructure
[3] MongoDB is a document-oriented database which comes with a simple query language. http://mongodb.org

5.3.2 Setting up a connection

Without specifying any connection parameters Leon connects to the following database:

- Host: 127.0.0.1
- Port: 27017
- Database: leon_test

To pass your own connection parameters you can use the `LeonMongoConfig` object. How to do this shows the following example:

```
var mongoConfig = new Packages.io.leon.persistence.mongo.LeonMongoConfig("mongo0.example.com",
 27017, "your_database");

install(new Packages.io.leon.persistence.mongo.LeonMongoModule(mongoConfig));
```

5.3.3 Working with MongoDB

If you have enabled Leon's MongoDB module, the MongoDB connection is bound to the server-side variable `leon.mongo`.

To access a Mongo collection within a server-side JavaScript file just use its name in the scope of `leon.mongo`. For example, the code to insert a document in a collection named `people` looks like this:

```
leon.mongo.people.insert({name: "John Doe"});
```

That's all! If the collection doesn't exist, MongoDB will create it for you.

To query a collection you can use the function `find`:

```
var cursor = leon.mongo.people.find({name: /^John.*$/});
cursor.forEach(function(person) {
  // do something with the person
});
```

The example above queries the collection `people` for documents with a field `name` starting with "John". The function `find` returns a cursor that can be used to iterate over the result.

Please see the MongoDB documentation [4] for more information about MongoDB functions. Generally speaking, all functions MongoDB provides can be accessed via Leon's Mongo module.

5.3.4 Using MongoDB from Java

TODO

5.4 SQL

Coming soon ...

[4] MongoDB Documentation http://www.mongodb.org/display/DOCS/Manual

CHAPTER

SIX

IDE SUPPORT

6.1 Leon Eclipse Plugin

For Leon we provide an Eclipse plugin including

- A Leon project wizard
- A project property page, where you can configure the Leon configuration file
- Content assist for the Leon configuration file when opened in the Eclipse JavaScript editor

6.1.1 System requirements

- Eclipse 3.7 or higher
- The `Eclipse IDE for Java Script Web Developers` package or another package including the `JavaScript Development Tools (org.eclipse.wst.jsdt.feature)` and `Java Development Tools (org.eclipse.jdt.feature)`
- Java 1.5 or higher

6.1.2 Leon project wizard

You can start the Leon project wizard from the common list of new project wizards under the category `Leon` or with the corresponding toolbar button. With these wizard you can configure the project location an the project set, the new project should be added to. While finishing the wizard, a Leon project with default JRE and java script libraries is created. A sample content for a Leon project including a sample configuration file is added too.

6.1.3 Leon project property page

For a Leon project you will find a `Leon` property page on the project properties dialog. On this page you can select the Leon configuration file (default: <project root>/config.js suitable for the sample content, created by the Leon wizard).

6.1.4 Content assist for the leon configuration file

If you open the Leon configuration file configured on the Leon property page in the Eclipse build-in java script editor, you have access to content proposals for methods later provided by the Leon framework. Please see *config* for detailed information about the provided methods.

INDEX

A

addLocation() (built-in function), 7

B

bind() (built-in function), 8
browser() (built-in function), 8

E

exposeUrl() (built-in function), 7

I

install() (built-in function), 7

L

leon.getLogger() (leon method), 7
loadFile() (built-in function), 8

S

setBaseDir() (built-in function), 7