jsass Documentation

Release 5.0.0

Tristan Lins

Jun 12, 2017

Contents

jsass is a Java sass compiler using libsass.

Contents:

CHAPTER 1

Basic examples

Compile file

Compiling a file is pretty simple, give an input file and an output file, the rest is just magic.

```
import static java.lang.System.err;
import static java.lang.System.out;

import io.bit3.jsass.CompilationException;
import io.bit3.jsass.Compiler;
import io.bit3.jsass.Options;
import io.bit3.jsass.Output;

import java.io.File;
import java.net.URI;

public class CompileFileExample {
  public static void main(String[] args) {
    URI inputFile = new File("stylesheet.scss").toURI();
    URI outputFile = new File("stylesheet.css").toURI();

    Compiler compiler = new Compiler();
    Options options = new Options();

    try {
      Output output = compiler.compileFile(inputFile, outputFile, options);

      out.println("Compiled successfully");
      out.println(output.getCss());
    } catch (CompilationException e) {
      err.println("Compile failed");
      err.println(e.getErrorText());
    }
  }
}
```

```
import static java.lang.System.err;
import static java.lang.System.out;

import io.bit3.jsass.CompilationException;
import io.bit3.jsass.Compiler;
import io.bit3.jsass.Options;
import io.bit3.jsass.Output;
import io.bit3.jsass.context.FileContext;

import java.io.File;
import java.net.URI;

public class CompileFileContextExample {
  public static void main(String[] args) {
    URI inputFile = new File("stylesheet.scss").toURI();
    URI outputFile = new File("stylesheet.css").toURI();

    Compiler compiler = new Compiler();
    Options options = new Options();

    try {
      FileContext context = new FileContext(inputFile, outputFile, options);
      Output output = compiler.compile(context);

      out.println("Compiled successfully");
      out.println(output.getCss());
    } catch (CompilationException e) {
      err.println("Compile failed");
      err.println(e.getErrorText());
    }
  }
}
```

Compile string

Compiling a string is pretty simple, give an input string, the rest is just magic. Providing an input file and output file is always a good idea. With this informations libsass can determine the default include path and calculate relative paths.

```
import static java.lang.System.err;
import static java.lang.System.out;

import io.bit3.jsass.CompilationException;
import io.bit3.jsass.Compiler;
import io.bit3.jsass.Options;
import io.bit3.jsass.Output;

import java.io.File;
import java.net.URI;

public class CompileStringExample {
  public static void main(String[] args) {
    String input = "body { color: red; }";
    URI inputFile = new File("stylesheet.scss").toURI();
    URI outputFile = new File("stylesheet.css").toURI();
```

```
17
18      Compiler compiler = new Compiler();
19      Options options = new Options();
20
21      try {
22        Output output = compiler.compileString(input, inputFile, outputFile, options);
23
24        out.println("Compiled successfully");
25        out.println(output.getCss());
26      } catch (CompilationException e) {
27        err.println("Compile failed");
28        err.println(e.getErrorText());
29      }
30    }
31  }
```

```
1   import static java.lang.System.err;
2   import static java.lang.System.out;
3
4   import io.bit3.jsass.CompilationException;
5   import io.bit3.jsass.Compiler;
6   import io.bit3.jsass.Options;
7   import io.bit3.jsass.Output;
8   import io.bit3.jsass.context.StringContext;
9
10  import java.io.File;
11  import java.net.URI;
12
13  public class CompileStringContextExample {
14    public static void main(String[] args) {
15      String input = "body { color: red; }";
16      URI inputFile = new File("stylesheet.scss").toURI();
17      URI outputFile = new File("stylesheet.css").toURI();
18
19      Compiler compiler = new Compiler();
20      Options options = new Options();
21
22      try {
23        StringContext context = new StringContext(input, inputFile, outputFile,␣
    ↪options);
24        Output output = compiler.compile(context);
25
26        out.println("Compiled successfully");
27        out.println(output.getCss());
28      } catch (CompilationException e) {
29        err.println("Compile failed");
30        err.println(e.getErrorText());
31      }
32    }
33  }
```

CHAPTER 2

Options

The options class allow to customize each compilation process. Most options are equals to the well known command line options from sass compilers.

Function providers

```
options.getFunctionProviders().add(new MyFunctions());
```

Headers

```
options.getHeaderImporters().add(new MyHeaderImporter());
```

Importers

```
options.getImporters().add(new MyImporter());
```

Include paths

```
options.getIncludePaths().add(new File("bower_components/foundation/scss");
```

Indention

```
options.setIndent("\t");
```

SASS syntax

Treat source_string as sass (as opposed to scss).

```
options.setIsIndentedSyntaxSrc(true);
```

Linefeed

```
options.setLinefeed("\r\n");
```

Omit source map url

Disable sourceMappingUrl in css output.

```
options.setOmitSourceMapUrl(true);
```

Output style

Output style for the generated css code.

```
options.setOutputStyle(io.bit3.jsass.OutputStyle.NESTED);
```

Precision

Precision for outputting fractional numbers.

```
options.setPrecision(6);
```

Inline source comments

If you want inline source comments.

```
options.setSourceComments(true);
```

Embed contents in source map

Embed include contents in maps.

```
options.setSourceMapContents(true);
```

Embedded source map

Embed sourceMappingUrl as data uri.

```
options.setSourceMapEmbed(true);
```

Source map

Path to source map file. Enables the source map generating. Used to create sourceMappingUrl.

```
options.setSourceMapFile(new File("stylesheet.css.map");
```

CHAPTER 3

Functions

libsass allow registration of custom functions. These functions are equivalent to `@function` functions in the sass language. jsass automatically maps methods from any java object to libsass and internally converts java values to libsass values and vise versa for you.

First you must write an object with public methods.

```
1  import io.bit3.jsass.annotation.Name;
2
3  public class MyFunctions {
4      public String hello(@Name("name") String name) {
5          return "Hello " + name;
6      }
7  }
```

Then register the object instance to the options.

```
options.getFunctionProviders().add(new MyFunctions());
```

jsass will map the method `MyFunctions::hello(name)` to libsass as `hello($name)`.

What methods are registered?

All directly declared public methods are registered as libsass functions. Non-public and inherited methods are not registered.

Special functions `@warn`, `@error`, `@debug`

Libsass allow to overwrite the `@warn`, `@error` and `@debug` functions. Simply mark the designated method with annotations, like this:

```
@WarnFunction
public void warn(String message) {
  logger.warn(message);
}

@ErrorFunction
public void error(String message) {
  logger.error(message);
}

@DebugFunction
public void debug(String message) {
  logger.debug(message);
}
```

Function signature

The function signature is build from the method name and the parameter annotation `@Name`. If the `@Name` annotation is missing, the name will be `argX`.

Default values

With the `@Default...Value` annotations, `@DefaultStringValue` for strings for example, you can set a default value. The default value is passed by libsass to your method. There is no way / need to use method overloading.

Value types

jsass brings all sass types as java types. If you prefer to use native java types, you can. jsass will convert the values for you as good as it can. For details have a look into the TypeUtils class, which will do the conversion.

Java to libsass

Java type	Libsass type	Notes
Primitive types		
SassNull	*null*	
null	*null*	
SassBoolean	boolean	
Boolean	boolean	
SassNumber	double	Unit depend on the SassNumber settings.
? extends Number	double	A number without any unit.
Complex types		
SassString	string	Quoting depend on the SassString settings.
String	string	Always quoted with double quotes.
? extends CharSequence	string	Always quoted with double quotes.
SassColor	color	
SassList	list	Separator depend on the SassList settings.
? extends Collection	list	always with comma separator
SassMap	map	
? extends Map	map	
Errors and warnings		
SassError	error	
SassWarning	warning	
? extends Throwable	error	If your function throw an exception or error, it will be returned to libsass as error value.

libsass to Java

Libsass type	Parameter type	resulting Java type	Notes
Primitive types			
null	*	*null*	Simply a *null* value!
boolean	SassBoolean	SassBoolean	
boolean	Boolean	Boolean	
double	SassNumber	SassNumber	
double	Number	SassNumber	
double	Double	Double	Unit get lost.
double	Float	Float	Unit and precision get lost.
double	Long	Long	Unit and fraction get lost.
double	Integer	Integer	Unit and fraction get lost.
double	Short	Short	Unit and fraction get lost.
double	Byte	Byte	Unit and fraction get lost.
Complex types			
string	SassString*	SassString*	
string	String	String	Quotation status get lost.
string	CharSequence	SassString*	
color	SassColor	SassColor	
list	SassList	SassList	
list	Collection	SassList	
map	Map<String, SassValue>	SassMap	

Note: Remind that `SassString implements CharSequence` which is incompatible with `java.lang.String`. If possible it is a good idea to use the `Sass*` type classes, but there is no need.

Note: Primitive types are also supported. jsass internally only use object types, but thanks to auto-boxing primitive type support is also provided.

CHAPTER 4

Importers

Importers are an experimental feature of libsass. They allow to manipulate the way how `@import` works.

> **Warning:** The import source string must be in SCSS syntax. SASS syntax is not supported yet!

In jsass importers must implement the `io.bit3.jsass.importer.Importer` interface.

```
import io.bit3.jsass.importer.Import;
import io.bit3.jsass.importer.Importer;

import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.Collection;
import java.util.Collections;

public class MyImporter implements Importer {
  @Override
  public Collection<Import> apply(String url, Import previous) {
    try {
      return Collections.singletonList(
          new Import(
              new URI("import.scss"),
              new File("public/assets/import.scss").toURI()
          )
      );
    } catch (URISyntaxException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Then register the object instance to the options.

```
options.getImporters().add(new MyImporter());
```

Thats all! From now on, each `@import` will be passed through your custom importer.

Skip importer

If you importer should be skipped, just return `null`.

```
public class MyImporter implements Importer {
    @Override
    public Collection<Import> apply(String url, Import previous) {
        // ...

        if (someReasonToSkipThisImporter) {
            return null;
        }

        // ...
    }
}
```

Skip import

Sometimes you may want to omit an `@import` directive. In this case, return an empty list.

```
public class MyImporter implements Importer {
    @Override
    public Collection<Import> apply(String url, Import previous) {
        // ...

        if (someReasonToSkipThisImportRule) {
            return new LinkedList<>();
        }

        // ...
    }
}
```

Import a file

Importing a file is one of the basic ways to import a source. Fill the `Import#importUri` with the relative file name and the `Import#absoluteUri` with the absolute file path, leave everything else empty. libsass will search the file in the path and import it.

```
Import fileImport = new Import(
        new URI("import.scss"),
        new File("public/assets/import.scss").toURI()
);
```

Import a string

Importing a string is as simple as importing a file. Just add the string contents to the import.

```
String contents = ".hello { content: 'Hello world'! }";

Import fileImport = new Import(
        new URI("import.scss"),
        new File("public/assets/import.scss").toURI(),
        contents
);
```

CHAPTER 5

License

CHAPTER 6

Changelog

Version 5.0.0

In jsass 5 the error handling has changed to full exceptions.

Before jsass 5:

```
Output output = compiler.compileFile(inputFile, outputFile, options);

if (0 == output.getErrorStatus()) {
  out.println("Compiled successfully");
  out.println(output.getCss());
} else {
  out.println("Compiled failed");
  out.println(output.getErrorText());
}
```

Since jsass 5:

```
try {
  Output output = compiler.compileFile(inputFile, outputFile, options);

  out.println("Compiled successfully");
  out.println(output.getCss());
} catch (CompilationException e) {
  out.println("Compiled failed");
  out.println(e.getErrorText());
}
```

Version 4.0.0

jsass 4+ using libsass 3.3+. The importer API has changed a little bit, so the jsass API changed too.

- `Import#uri` has been renamed to `Import#importUri`
- `Import#base` has been renamed to `Import#absoluteUri`

Here is an example what is the effect of the change:

```
// jsass 3+ style
Import fileImport = new Import(
        new URI("import.scss"),
        new File("public/assets").toURI(),
        contents
);
```

```
// jsass 4+ style
Import fileImport = new Import(
        new URI("import.scss"),
        new File("public/assets/import.scss").toURI(),
        contents
);
```

As you can see you must provide the absolute file path as second argument now, instead of the absolute base directory path.

CHAPTER 7

Internals

jsass use an JNI adapter written in C to bind libsass to java. You can find the sources in our repository .

Binaries are generally provided for Linux x86_64, Mac OS X and Windows 64bit. If binaries are missing they where deleted due to changes on the jni code. Feel free to rebuild them an make a PR with the new binaries. You can use one of our `bin/make-*` scripts to build the binaries.