

( / )

# Templating with Handlebars

Last modified: February 9, 2020

by baeldung (<https://www.baeldung.com/author/baeldung/>)

Java (<https://www.baeldung.com/category/java/>) +

Java String (<https://www.baeldung.com/tag/java-string/>)

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring and Spring Boot.

>> CHECK

X

## 1. Overview

In this tutorial, we'll look into the Handlebars.java (<https://jknack.github.io/handlebars.java/>) library for easy template management.

## 2. Maven Dependencies

Let's start with adding the *handlebars* (<https://search.maven.org/search?q=com.github.jknack%2Ba:handlebars>) dependency:

```
1 <dependency>
2   <groupId>com.github.jknack</groupId>
3   <artifactId>handlebars</artifactId>
4   <version>4.1.2</version>
5 </dependency>
```

## 3. A Simple Template

We use cookies to improve your experience with the site. To find out more, you can read the full [Privacy and Cookie Policy](#) ([privacy-policy](#)).

A Handlebars template can be any kind of text file. It consists of tags like `{{name}}` and `{{#each people}}`.

OK



```

1 public class Person {
2     private String name;
3     private boolean busy;
4     private Address address = new Address();
5     private List<Person> friends = new ArrayList<>();
6
7     public static class Address {
8         private String street;
9     }
10 }

```

Using the *Person* class, we'll achieve the same result as the previous example:

```

1 @Test
2 public void whenParameterObjectIsSupplied_ThenDisplays() throws IOException {
3     Handlebars handlebars = new Handlebars();
4     Template template = handlebars.compileInline("Hi {{name}}!");
5     Person person = new Person();
6     person.setName("Baeldung");
7
8     String templateString = template.apply(person);
9
10    assertThat(templateString).isEqualTo("Hi Baeldung!");
11 }

```

*{{name}}* in our template will drill into our *Person* object and get the value of the *name* field.

## 4. Template Loaders



So far, we've  
read templates

can also

Handlebars.java provides special support for reading templates from the classpath, filesystem or servlet context. By default, Handlebars scans the classpath to load the given template:

```

1 @Test
2 public void whenNoLoaderIsGiven_ThenSearchesClasspath() throws IOException {
3     Handlebars handlebars = new Handlebars();
4     Template template = handlebars.compile("greeting");
5     Person person = getPerson("Baeldung");
6
7     String templateString = template.apply(person);
8
9     assertThat(templateString).isEqualTo("Hi Baeldung!");
10 }

```

So, because we called *compile* instead of *compileInline*, this is a hint to Handlebars to look for */greeting.hbs* on the classpath.

However, we can also configure these properties with *ClassPathTemplateLoader*:

```

1 @Test
2 public void whenClasspathTemplateLoaderIsGiven_ThenSearchesClasspathWithPrefixSuffix() throws
3     IOException {
4     TemplateLoader loader = new ClassPathTemplateLoader("/handlebars", ".html");
5     Handlebars handlebars = new Handlebars(loader);
6     Template template = handlebars.compile("greeting");
7     // ... same as before
8 }

```

In this case, we're telling Handlebars to look for the */handlebars/greeting.html* on the classpath.

Finally, we can chain multiple *TemplateLoader* instances:

## 6. Custom Template Helpers

We can also create our own custom helpers.

### 6.1. Helper

The *Helper* interface enables us to create a template helper.

As the first step, we must provide an implementation of *Helper*.

```
1 new Helper<Person>() {  
2     @Override  
3     public Object apply(Person context, Options options) throws IOException {  
4         String busyString = context.isBusy() ? "busy" : "available";  
5         return context.getName() + " - " + busyString;  
6     }  
7 }
```

As we can see, the *Helper* interface has only one method which accepts the *context* and *options* objects. For our purposes, we'll output the *name* and *busy* fields of *Person*.

After creating the helper, we must also register our custom helper with Handlebars:

```
1 @Test  
2 public void whenHelperIsCreated_ThenCanRegister() throws IOException {  
3     Handlebars handlebars = new Handlebars(templateLoader);  
4     handlebars.registerHelper("isBusy", new Helper<Person>(){  
5  
6  
7  
8  
9         }  
10    });  
11  
12    // implementation details  
13 }
```

In our example, we're registering our helper under the name of *isBusy* using the *Handlebars.registerHelper()* method.

As the last step, we must define a tag in our template using the name of the helper:

```
1 {{#isBusy this}}{{/isBusy}}
```

Notice that each helper has a starting and ending tag.

### 6.2. Helper Methods

When we use the *Helper* interface, we can only create only one helper. In contrast, a helper source class enables us to define multiple template helpers.

Moreover, we don't need to implement any specific interface. We just write our helper methods in a class then HandleBars extracts helper definitions using reflection:

```

1 public class HelperSource {
2
3     public String isBusy(Person context) {
4         String busyString = context.isBusy() ? "busy" : "available";
5         return context.getName() + " - " + busyString;
6     }
7
8     // Other helper methods
9 }

```

Since a help  
helper regis

single

```

1 @Test
2 public void whenHelperSourceIsCreated_ThenCanRegister() throws IOException {
3     Handlebars handlebars = new Handlebars(templateLoader);
4     handlebars.registerHelpers(new HelperSource());
5
6     // Implementation details
7 }

```

We're registering our helpers using the `Handlebars.registerHelpers()` method. Moreover, the name of the helper method becomes the name of the helper tag.

## 7. Template Reuse

The Handlebars library provides several ways to reuse our existing templates.

### 7.1. Template Inclusion

Template inclusion is one of the approaches for reusing templates. It favors the composition of the templates.

```
1 <h4>Hi {{name}}!</h4>
```

This is the content of the *header* template – *header.html*.

In order to use it in another template, we must refer to the *header* template.

```

1 {{>header}}
2 <p>This is the page {{name}}</p>

```

We have the page template – *page.html* – which includes the *header* template. [Privacy and Cookie Policy \(/privacy-policy\)](#)

When Handlebars.java processes the template, the final output will also contain the contents of *header*.

Ok



```
1  @Test
2  public void whenOtherTemplateIsReferenced_ThenCanReuse() throws IOException {
3      Handlebars handlebars = new Handlebars(templateLoader);
4      Template template = handlebars.compile("page");
5      Person person = new Person();
6      person.setName("Baeldung");
7
8      String templateString = template.apply(person);
9
10     assertThat(templateString)
11         .contains("<h4>Hi Baeldung!</h4>", "<p>This is the page Baeldung</p>");
12 }
```

## 7.2. Template Inheritance

Alternatively to composition, Handlebars provides the template inheritance.

We can achieve inheritance relationships using the `[[#block]]` and `[[#partial]]` tags:

```
1  <html>
2  <body>
3      {{#block "intro"}}
4          This is the intro
5      {{/block}}
6      {{#block "message"}}
7      {{/block}}
8  </body>
9  </html>
```

By doing so



To apply inheritance, we need to override these *blocks* in other templates using `[[#partial]]`:

```
1  {{#partial "message" }}
2      Hi there!
3  {{/partial}}
4  {{> messagebase}}
```

This is the *simplemessage* template. Notice that we're including the *messagebase* template and also overriding the *message* block.

## 8. Summary

In this tutorial, we've looked at Handlebars.java to create and manage templates.

We started with the basic tag usage and then looked at the different options to load the Handlebars templates.

We also investigated the template helpers which provide a great deal of functionality. Lastly, we looked at the different ways to reuse our templates.

Finally, check out the source code for all examples over on [GitHub](https://github.com/eugenp/tutorials/tree/master/libraries-2) (https://github.com/eugenp/tutorials/tree/master/libraries-2).

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE ([/!s-course-end](#))



Learning to "Build your API  
with Spring"?

>> Get the eBook

Comments are closed on this article!

We use cookies to improve your experience with the site. To find out more, you can read the full [Privacy and Cookie Policy](#) ([/privacy-policy](#)).

Ok



 **ezoic** (<https://www.ezoic.com/what-is-ezoic/>)

[report this ad](#)

## CATEGORIES

[SPRING \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/\)](https://www.baeldung.com/category/spring/)  
[REST \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/\)](https://www.baeldung.com/category/rest/)  
[JAVA \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/\)](https://www.baeldung.com/category/java/)  
[SECURITY \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/\)](https://www.baeldung.com/category/security-2/)  
[PERSISTENCE \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/\)](https://www.baeldung.com/category/persistence/)  
[JACKSON \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/\)](https://www.baeldung.com/category/json/jackson/)  
[HTTP CLIENT-SIDE \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/\)](https://www.baeldung.com/category/http/)  
[KOTLIN \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/\)](https://www.baeldung.com/category/kotlin/)

## SERIES

[JAVA "BACK TO BASICS" TUTORIAL \(/JAVA-TUTORIAL\)](/java-tutorial/)  
[JACKSON JSON TUTORIAL \(/JACKSON\)](/jackson/)  
[HTTPCLIENT 4 TUTORIAL \(/HTTPCLIENT-GUIDE\)](/httpclient-guide/)  
[REST WITH SPRING TUTORIAL \(/REST-WITH-SPRING-SERIES\)](/rest-with-spring-series/)  
[SPRING PERSISTENCE TUTORIAL \(/PERSISTENCE-WITH-SPRING-SERIES\)](/persistence-with-spring-series/)  
[SECURITY WITH SPRING \(/SECURITY-SPRING\)](/security-spring/)

## ABOUT

[ABOUT BAELDUNG \(/ABOUT\)](/about/)  
[THE COURSES \(HTTPS://COURSES.BAELDUNG.COM\)](https://courses.baeldung.com)  
[JOBS \(/TAG/ACTIVE-JOB/\)](/tag/active-job/)  
[THE FULL ARCHIVE \(/FULL-ARCHIVE\)](/full-archive/)  
[WRITE FOR BAELDUNG \(/CONTRIBUTION-GUIDELINES\)](/contribution-guidelines/)  
[EDITORS \(/EDITORS\)](/editors/)  
[OUR PARTNERS \(/PARTNERS\)](/partners/)  
[ADVERTISE ON BAELDUNG \(/ADVERTISE\)](/advertise/)

[TERMS OF SERVICE \(/TERMS-OF-SERVICE\)](/terms-of-service/)

[PRIVACY POLICY \(/PRIVACY-POLICY\)](/privacy-policy/)

[COMPANY INFO \(/BAELDUNG-COMPANY-INFO\)](/baeldung-company-info/)

[CONTACT \(/CONTACT\)](/contact/)

We use cookies to improve your experience with the site. To find out more, you can read the full [Privacy and Cookie Policy \(/privacy-policy\)](/privacy-policy/).

Ok



We use cookies to improve your experience with the site. To find out more, you can read the full [Privacy and Cookie Policy](#).[\(privacy-policy\)](#)

Ok



Then we fill in these tags by passing a context object, like a *Map* or other *Object*.

### 3.1. Using *this*

To pass a single *String* value to our template, we can use any *Object* as the context. We must also use the `{{this}}` tag in our template.

Then Handlebars calls the *toString* method on the context object and replaces the tag with the result:

```
1 @Test
2 public void whenThereIsNoTemplateFile_ThenCompilesInline() throws IOException {
3     Handlebars handlebars = new Handlebars();
4     Template template = handlebars.compileInline("Hi {{this}}!");
5
6     String templateString = template.apply("Baeldung");
7
8     assertThat(templateString).isEqualTo("Hi Baeldung!");
9 }
```

In the above example, we first create an instance of *Handlebars*, our API entry point.

Then, we give that instance our template. Here, we just pass the template inline, but we'll see in a moment some more powerful ways.

Finally, we give the compiled template our context. `{{this}}` is just going to end up calling *toString*, which is why we see *"Hi Baeldung!"*.

---

### 3.2. Passi

We just saw

```
1 @Test
2 public void whenParameterMapIsSupplied_thenDisplays() throws IOException {
3     Handlebars handlebars = new Handlebars();
4     Template template = handlebars.compileInline("Hi {{name}}!");
5     Map<String, String> parameterMap = new HashMap<>();
6     parameterMap.put("name", "Baeldung");
7
8     String templateString = template.apply(parameterMap);
9
10    assertThat(templateString).isEqualTo("Hi Baeldung!");
11 }
```

Similar to the previous example, we're compiling our template and then passing the context object, but this time as a *Map*.

Also, notice that we're using `{{name}}` instead of `{{this}}`. This means that our map must contain the key, *name*.

---

### 3.3. Passing a Custom Object as Context Object

We use cookies to improve your experience with the site. To find out more, you can read the full [Privacy and Cookie Policy](#) ([privacy-policy](#)).

Ok

We can also pass a custom object to our template:



```

1  @Test
2  public void whenMultipleLoadersAreGiven_ThenSearchesSequentially() throws IOException {
3      TemplateLoader firstLoader = new ClassPathTemplateLoader("/handlebars", ".html");
4      TemplateLoader secondLoader = new ClassPathTemplateLoader("/templates", ".html");
5      Handlebars handlebars = new Handlebars().with(firstLoader, secondLoader);
6      // ... same as before
7  }

```

So, here, we've got two loaders, and that means Handlebars will search two directories for the *greeting* template.

## 5. Built-in Helpers

Built-in helpers provide us additional functionality when writing our templates.



### 5.1. *with* Helper

The *with* helper changes the current context:

```

1  {{#with address}}
2  <h4>I live in {{street}}</h4>
3  {{/with}}

```

In our sample template, the *{{#with address}}* tag starts the section and the *{{/with}}* tag ends it.

In essence, we're drilling into the current context object – let's say *person* – and setting *address* as the local context for the *with* section. Thereafter, every field reference in this section will be prepended by *person.address*.

So, the *{{street}}* tag will hold the value of *person.address.street*.

```

1  @Test
2  public void whenUsedWith_ThenContextChanges() throws IOException {
3      Handlebars handlebars = new Handlebars(templateLoader);
4      Template template = handlebars.compile("with");
5      Person person = getPerson("Baeldung");
6      person.getAddress().setStreet("World");
7
8      String templateString = template.apply(person);
9
10     assertThat(templateString).contains("<h4>I live in World</h4>");
11 }

```

We're compiling our template and assigning a *Person* instance as the context object. Notice that the *Person* class has an *Address* field. This is the field we're supplying to the *with* helper.

Though we went one level into our context object, it is perfectly fine to go deeper if the context object has several nested levels.

We use cookies to improve your experience with the site. To find out more, you can read the full [Privacy and Cookie Policy](#) ([privacy-policy](#)).

### 5.2. *each* Helper

Ok

The *each* helper iterates over a collection:



```
1  {{#each friends}}
2  <span>{{name}} is my friend.</span>
3  {{/each}}
```

As a result of starting and closing the iteration section with *{{#each friends}}* and *{{/each}}* tags, Handlebars will iterate over the *friends* field of the context object.

```
1  @Test
2  public void whenUsedEach_ThenIterates() throws IOException {
3      Handlebars handlebars = new Handlebars(templateLoader);
4      Template template = handlebars.compile("each");
5      Person person = getPerson("Baeldung");
6      Person friend1 = getPerson("Java");
7      Person friend2 = getPerson("Spring");
8      person.getFriends().add(friend1);
9      person.getFriends().add(friend2);
10
11     String templateString = template.apply(person);
12
13     assertThat(templateString)
14         .contains("<span>Java is my friend.</span>", "<span>Spring is my friend.</span>");
15 }
```

In the example, we're assigning two *Person* instances to the *friends* field of the context object. So, Handlebars repeats the HTML part two times in the final output.

### 5.3. if Helper



Lastly, the *if* helper

```
1  {{#if busy}}
2  <h4>{{name}} is busy.</h4>
3  {{else}}
4  <h4>{{name}} is not busy.</h4>
5  {{/if}}
```

In our template, we're providing different messages according to the *busy* field.

```
1  @Test
2  public void whenUsedIf_ThenPutsCondition() throws IOException {
3      Handlebars handlebars = new Handlebars(templateLoader);
4      Template template = handlebars.compile("if");
5      Person person = getPerson("Baeldung");
6      person.setBusy(true);
7
8      String templateString = template.apply(person);
9
10     assertThat(templateString).contains("<h4>Baeldung is busy.</h4>");
11 }
```

We use cookies to improve your experience with the site. To find out more, you can read the full [Privacy and Cookie Policy \(/privacy-policy\)](#).  
After compiling the template, we're setting the context object. Since the *busy* field is *true*, the final output becomes *<h4>Baeldung is busy.</h4>*.

Ok