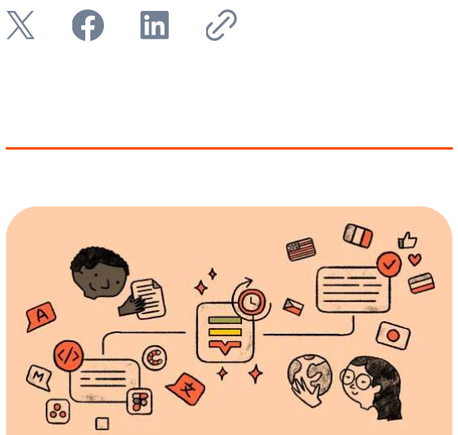
Tutorials

Spring Boot internationalization i18n: Step-by-step with examples

Ilya Krukowski, Updated on April 6, 2023 • 9 min read



A short time ago we looked into the basics of Java i18n. In this article, let's take a step into the web application realm and see how the Spring Boot framework handles internationalization (i18n).

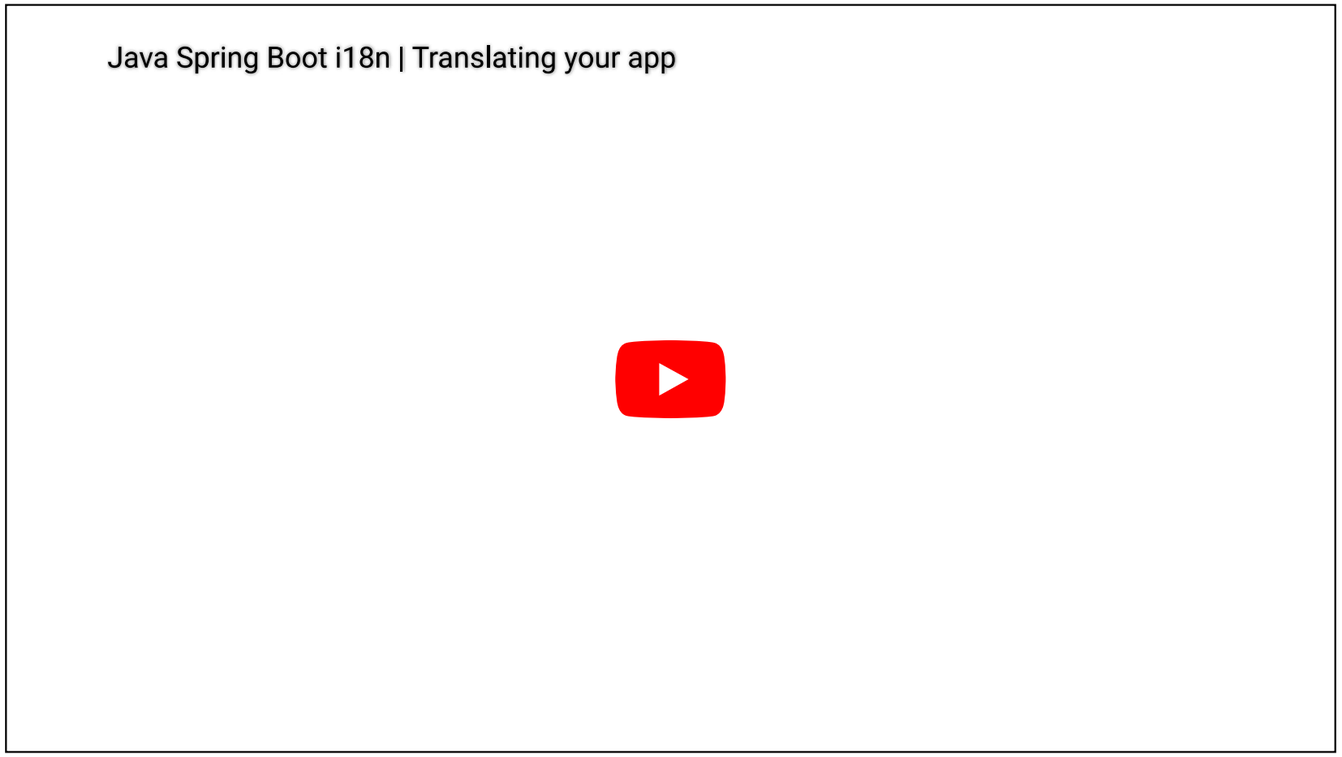
When developing a web application, we tend to code it using a collection of the most efficient, the most popular, and the most sought-after programming languages for both our front end and back end. But what about spoken languages? Most of the time, with or without our knowledge, we depend on the built-in translation engines of our customers' browsers to handle the required translations. Don't we?

In the ever-globalizing world we live in, we need our web applications to reach as wide an audience as possible. Here enters the much-required concept of *internationalization*. In this article, we will be looking at how i18n works on the popular Spring Boot framework.

We will be covering the following topics in this tutorial:

- I18n internationalization on Spring Boot.
- `MessageSource` interface and its uses.
- Locale resolving through `LocaleResolver`, `LocaleChangeInterceptor` classes.
- Storing the user-preferred locale in cookies.
- Switching between languages.
- Pluralization with the help of ICU4J standards.
- Date-time localization using `@DateTimeFormat` annotation.

The source code for this article is available on GitHub. Find a working demo and a playground on GitPod.



Prerequisites

- Spring Boot 3+ (though most of the described concepts should be applicable to version 2 as well).
- Java SDK 17+
- Maven 3.3+

I18n on Spring Boot

First off, let us create a simple Spring Boot example project using Maven to get a grasp of how internationalization works on Spring.

Let's go ahead and make a new Spring Boot application named `javai18nspringboot`. To achieve this, head over to Spring Initializr and generate a new Spring Boot project with the following set up:

```
Group:      com.lokalise
Artifact:   javai18nspringboot
Packaging:  Jar
Java:       17
```

Save the generated ZIP file and extract it to a local directory of your choice. Next, simply start your favorite IDE and open the extracted `javai18nspringboot` project.

Adding dependencies

Open up the `pom.xml` file in the project root and add the `spring-boot-starter-web` dependency as well as the Thymeleaf templating engine to the `<dependencies>` tag:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

If you are using Gradle, `dependencies` section in your `build.gradle` file should look like this:

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter'
  implementation 'org.springframework.boot:spring-boot-starter-web'
  implementation 'org.springframework.boot:spring-boot-starter-
thymeleaf'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Nice!

Adding translation files

Next, we need to add some language resources to our app. In the project `src/main/resources`, create a new `lang` directory. Create a simple Java properties file `messages.properties` inside the newly created directory, and add a few keys and translation values:

```
hello=Hello!
welcome=Welcome to my app
switch-en=Switch to English
switch-it=Switch to Italian
```

`messages` will act as the base name for our set of translations but you can use any other name.

`messages.properties` is the default translation file that our Spring Boot application will resort to in the case that no match was found.

Secondly, let's add another `messages_it.properties` file to the same `lang` directory to hold localization resource data for the Italian locale. Duplicate the same keys of the default resource file on `messages_it.properties` file. As for the values, add the corresponding Italian translations:

```
hello=Ciao!
welcome=Benvenuti nella mia app
switch-en=Passa all'Inglese
switch-it=Passa all'italiano
```

You can also open the `src/main/resources/application.properties` file and configure the base name:

```
spring.messages.basename=lang/messages
```

Translation files naming rules

When naming translation files you'll have to follow the same naming rules as used by Java's built-in i18n functions when naming language resource files:

- All resource files must reside in the same package.

- All resource files must share a common **base name**.
- The default resource file should simply have the base name (for example `messages.properties` or `res.properties`)
- Additional resource files must be named following this pattern: `BASENAME_LOCALE` (for example, `messages_it.properties`).
- Regional suffixes are supported as well: `BASENAME_LOCALE_REGIONAL` (for example, `messages_en_US.properties`).

Testing it out

Let's see how `ResourceBundleMessageSource` works. Open the `src/main/java/com/lokalise/javai18nspringboot/JavaI18nSpringBoot.java` file and add the following code:

```
// other imports ...
import org.springframework.context.support.ResourceBundleMessageSource;
import java.util.Locale;

@SpringBootApplication
public class Javai18nspringbootApplication {
  public static void main(String[] args) {
    ResourceBundleMessageSource messageSource = new
ResourceBundleMessageSource();
    messageSource.setBasenames("lang/messages");
    messageSource.setDefaultEncoding("UTF-8");
    System.out.println(messageSource.getMessage("hello", null,
Locale.ITALIAN));
    SpringApplication.run(Javai18nspringbootApplication.class, args);
  }
}
```

Now you can start the app by running the following command:

```
mvnw spring-boot:run --quiet
```

Navigate to `localhost:8080` and make sure that the "Ciao!" text is printed to the terminal. You'll probably see various warning messages but we'll fix those later.

Spring Boot web application

Thanks to the magic of Spring Boot we have already completed building the skeleton of our Spring Boot internationalization example project. Now, it is time to give it some i18n functionalities.

Meet LocaleResolver

The LocaleResolver interface deals with locale resolution required when localizing web applications to specific locales. Spring aptly ships with a few LocaleResolver implementations that may come in handy in various scenarios:

- `FixedLocaleResolver` — always resolves the locale to a singular fixed language mentioned in the project properties. Mostly used for debugging purposes.
- `AcceptHeaderLocaleResolver` — resolves the locale using an "accept-language" HTTP header retrieved from an HTTP request.
- `SessionLocaleResolver` — resolves the locale and stores it in the HttpSession of the user. But as you might have wondered, yes, the resolved locale data is persisted only for as long as the session is live.
- `CookieLocaleResolver` — resolves the locale and stores it in a cookie stored on the user's machine. Now, as long as browser cookies aren't cleared by the user, once resolved the resolved locale data will last even between sessions. Cookies save the day!

Use a CookieLocaleResolver

Create a new file `MyBeansConfig.java` inside the `src/main/java/com/lokalise/javai18nspringboot` folder:

```java
package com.lokalise.javai18nspringboot;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.i18n.CookieLocaleResolver;
import java.util.Locale;
import java.util.TimeZone;

@Configuration
public class MyBeansConfig {

  @Bean
  public LocaleResolver localeResolver() {
      CookieLocaleResolver localeResolver = new CookieLocaleResolver();
      localeResolver.setDefaultLocale(Locale.ENGLISH);
      localeResolver.setDefaultTimeZone(TimeZone.getTimeZone("UTC"));

      return localeResolver;
  }
}
```

`LocaleResolver` interface is implemented using Spring's built-in `CookieLocaleResolver` implementation. We also set the default locale and the timezone here.

Add a LocaleChangeInterceptor

Okay, now our application knows how to resolve and store locales. However, when users from different locales visit our app, who's going to switch the application's locale accordingly? Or in other words, how do we localize our web application to the specific locales it supports?

For this, we'll add an interceptor bean that will intercept each request that the application receives, and eagerly check for a `localeData` parameter on the HTTP request. If found, the interceptor uses the `localeResolver` we coded earlier to register the locale it found as the current user's locale. Let's add the following code to the same `MyBeansConfig.java` file:

```
// other imports ...
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;

// other code ...

@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
  LocaleChangeInterceptor localeChangeInterceptor = new
LocaleChangeInterceptor();
  localeChangeInterceptor.setParamName("localeData");

  return localeChangeInterceptor;
}
```

Now, to make sure this interceptor properly intercepts all incoming requests, we should add it to the Spring InterceptorRegistry.

Open the `JavaI18nSpringBoot.java` file in the same folder and add the following code:

```
// other imports ...
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import
org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;

@SpringBootApplication
public class Javai18nspringbootApplication implements WebMvcConfigurer {
// <--- modify this line!
  private final LocaleChangeInterceptor localeChangeInterceptor;
```

```
    public Javai18nspringbootApplication(LocaleChangeInterceptor
localeChangeInterceptor) {
      this.localeChangeInterceptor = localeChangeInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry interceptorRegistry) {
      interceptorRegistry.addInterceptor(localeChangeInterceptor);
    }

    public static void main(String[] args) {
                // ...
    }

  }
```

Create a controller

Create a new `HelloController.java` file within the same directory:

```
package com.lokalise.javai18nspringboot;
import org.springframework.stereotype.Controller;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.time.LocalDate;
import java.time.LocalDateTime;

@Controller public class HelloController {

}
```

Now, let's add a GET mapping to the root URL. Add the following to the `HelloController.java` file:

```
// imports ...

@Controller public class HelloController {
  @GetMapping("/")
  public String hello() {
      return "hello";
  }
}
```

Implement a view

Next, it's time to create a simple view on our application. Create the `src/main/resources/templates` directory and add a new `hello.html` file inside:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"> <!-- 1 -->
<head>
    <meta charset="UTF-8">
    <title th:text="#{welcome}"></title> <!-- 2 -->
 </head>
 <body>
 <span th:text="#{hello}"></span><br>
 <span th:text="#{welcome}"></span><br>
 <button type="button" th:text="#{switch-en}"
onclick="window.location.href='http://localhost:8080/?localeData=en'">
</button> <!-- 3 -->
 <button type="button" th:text="#{switch-it}"
onclick="window.location.href='http://localhost:8080/?localeData=it'">
</button>
 </body>
 </html>
```

Main things to note:

1. We declare Thymeleaf namespace in order to support `th:*` attributes.

2. Fetch the translations using the `th:text`.
3. Add the locale switchers. Upon clicking the button, the page is reloaded with an additional `localeData=it` (or `en`) parameter. This in turn causes our `LocaleChangeInterceptor` to kick in and resolve the template in the chosen language.

Test functionality

Let's see if our Spring Boot application correctly performs internationalization. Run the project, navigate to `localhost:8080` and click on the language switching buttons to make sure the text is translated properly.

As a nifty bonus, switch to one locale, close and reopen the browser, and navigate to the root URL again; since we used CookieLocaleResolver as our `LocaleResolver` implementation, you'll see that the chosen locale choice has been retained.

Scour Spring Boot i18n

Let's skim through a few more features that could turn out to be useful when internationalizing our Spring Boot application.

Pluralization

With the internationalization of our Spring Boot app aiming to support various locales, pluralization can become a somewhat overlooked, yet crucial step.

To demonstrate the point, let's suppose we need to handle text representing some apples based on a provided quantity. So, for the English language, it would take this form:

- 0 apples
- 1 apple
- 2 apples

In order to handle pluralization, we can take the help of the spring-icu library which introduces ICU4J message formatting features into Spring. Open the `pom.xml` file and add a new dependency:

```
<dependency>
  <groupId>com.github.transferwise</groupId>
  <artifactId>spring-icu</artifactId>
  <version>0.3.0</version>
</dependency>
```

Also make sure to add `repositories` before the `dependencies` section:

```
<repositories>
  <repository>
    <id>jitpack.io</id>
    <url>https://jitpack.io</url>
  </repository>
</repositories>
```

For Gradle, you would use the following code:

```
repositories {
  mavenCentral()
  maven { url 'https://jitpack.io' }
}

dependencies {
    implementation 'com.github.transferwise:spring-icu:0.3.0'
}
```

Open the `MyBeansConfig.java` file and add new imports:

```
import com.transferwise.icu.ICUMessageSource;
```

```
import com.transferwise.icu.ICUReloadableResourceBundleMessageSource;
```

Also add the `ICUMessageSource` bean. Make sure to set its base name correctly with a `classpath:` prefix, like so:

```
@Bean
public ICUMessageSource messageSource() {
  ICUReloadableResourceBundleMessageSource messageSource = new
ICUReloadableResourceBundleMessageSource();
  messageSource.setBasename("classpath:lang/messages");
  return messageSource;
}
```

Secondly, add a `plural` property to the `messages.properties` file indicating how to deal with particular quantities of apples:

```
plural={0} {0, plural, zero{apples}one{apple}other{apples}}
```

Note that this follows the `FormatElement: { ArgumentIndex , FormatType , FormatStyle }` pattern mentioned on MessageFormat with a `'plural' FormatType` added by the `spring-icu` library.

Let's also add Italian translation:

```
plural={0} {0, plural, zero{mele}one{mela}other{mele}}
```

Finally, add these lines to the `hello.html` template:

```
<br><span th:text="#{plural(0)}"></span>
<br><span th:text="#{plural(1)}"></span>
```

```
<br><span th:text="#{plural(22)}"></span>
```

Date and time

We can use the @DateTimeFormat Spring annotation to parse – or in other terms, deserialize – a String date-time input into a LocalDate or LocalDateTime object.

Open up the `HelloController.java` file and add a new GET mapping:

```
@GetMapping("/datetime")
@ResponseBody
public String dateTime(@RequestParam("date") @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE) LocalDate date,
                       @RequestParam("datetime") @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE_TIME) LocalDateTime datetime) {
    return date.toString() + "<br>" + datetime.toString();
}
```

Run the application and call the `/datetime` GET endpoint passing parameters as follows:

```
http://localhost:8080/datetime?date=1993-04-25&datetime=2018-11-
22T01:30:00.000-05:00
```

The `date` param contains a string in the most common ISO Date Format (`yyyy-MM-dd`). The `datetime` param is a string in the most common ISO DateTime Format (`yyyy-MM-dd'T'HH:mm:ss.SSSXXX`).

Lokalise to the rescue

By now you must be thinking...

Wow, okay I get it. This is an invaluable task that my web app will require to reach my expected audience. But isn't there an easier way to get all this done?

Meet Lokalise, the translation management system that takes care of all your Spring Boot app's internationalization needs. With features like:

- Easy integration with various other services
- Collaborative translations
- Quality assurance tools for translations
- Easy management of your translations through a central dashboard

Plus, loads of others, Lokalise will make your life a whole lot easier by letting you expand your web application to all the locales you'll ever plan to reach.

Get started with Lokalise in just a few steps:

- Sign up for a free trial (no credit card information required).
- Log in to your account.
- Create a new project under any name you like.
- Upload your translation files and edit them as required.

That's it! You have already completed the baby steps to Lokalise-*ing* your web application. See the Getting Started section for a collection of articles that will give all the help you'll need to kick-start the Lokalise journey. Also, refer Lokalise API Documentation for a complete list of REST commands you can call on your Lokalise translation project.

Conclusion

Huge thanks to my colleagues Anton Malich for helping to write this article.

In conclusion, in this tutorial we looked into how we can localize to several locales and integrate internationalization into a Spring Boot project. We learned how to perform simple translations using `MessageSource` implementations, use `LocaleResolver`, `LocaleChangeInterceptor` classes to resolve languages using the details of incoming HTTP requests, and how we can switch to a different language at the click of a button in our internationalized Spring Boot web application.

Additionally, we reviewed ways to manage pluralization of values, localize date and time, conduct language switching, and store the chosen language on a Spring Boot web application.

Further reading

- Java internationalization: Learn the basics
- Java LocalDate localization tutorial: step by step examples
- How to choose the best translation management system for your team and company
- Date and time localization
- I18n and I10n: List of developer tutorials

Tutorials

Author

Ilya Krukowski
Lead of content, SDK/integrations dev

Ilya is a lead of content/documentation/onboarding at Lokalise, an IT tutor and author, web developer, and ex-Microsoft/Cisco specialist. His primary programming languages are Ruby, JavaScript, Python, and Elixir. He enjoys coding, teaching people and learning new things. In his free time he writes educational posts, participates in OpenSource projects, tweets, goes in for sports and plays music.

Related articles

Guides · Localization

How to reach new audiences by localizing your HubSpot content

Localized content gets 12 times more engagement than non-localized content. It just goes to show that people feel more comfortable consuming content in their preferred language. And, it's a VERY...

Updated on June 21, 2024 · Rachel Wolff

General

Mobile content management system: what is it and why is it important

With over 6.4 billion smartphone users and five million apps, a great mobile app can be a differentiator — but it's not the distinction it once was. To stand out...

Updated on June 14, 2024 · Rachel Wolff

Tutorials

Libraries and frameworks to translate JavaScript apps

In our previous discussions, we delved into localization strategies for backend frameworks, particularly Rails and Phoenix. Today, we shift our focus to the front-end, and talk on how to translate...

Updated on June 3, 2024 · Ilya Krukowski

Case studies

Behind the scenes of localization with one of Europe's leading digital health providers

Read more →