Seam International Module

Reference Guide

3.1.0.Beta4

by Ken Finnigan and Lincoln Baxter III

Introduction v
1. Installation 1
2. Timezones 3
2.1. Joda Time 3
2.2. Application TimeZone 3
2.3. User TimeZone 4
2.4. Available TimeZones 4
3. Locales 7
3.1. Application Locale 7
3.2. User Locale 7
3.3. Available Locales 8
4. Messages 9
4.1. Message Creation 9
4.2. Properties Files 10

Introduction

The goal of Seam International is to provide a unified approach to configuring locale, timezone and language. With features such as Status message propagation to UI, multiple property storage implementations and more.

Installation

Most features of Seam International are installed automatically by including `seam-international.jar` in the web application library folder. If you are using *Maven* [http://maven.apache.org/] as your build tool, you can add the following dependency to your `pom.xml` file:

```
<dependency>
   <groupId>org.jboss.seam.international</groupId>
   <artifactId>seam-international</artifactId>
   <version>${seam-international-version}</version>
</dependency>
```

Tip

Replace ${seam-international-version} with the most recent or appropriate version of Seam International.

Timezones

To support a more developer friendly way of handling TimeZones, in addition to supporting JDK `TimeZone`, we have added support for using Joda-Time through their `DateTimeZone` class. Don't worry, it provides convenience methods for converting to JDK `TimeZone`.

2.1. Joda Time

To activate Joda-Time for i18n within your project you will need to add the following Maven dependency:

```
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>1.6</version>
  </dependency>
```

2.2. Application TimeZone

We have an Application time zone that is available with Joda-Time (`DateTimeZone`) or the JDK (`TimeZone`) that can be retrieved with

```
@Inject
private DateTimeZone applicationDateTimeZone;

@Inject
private TimeZone applicationTimeZone
```

It can also be accessed through EL by the name "defaultDateTimeZone" for Joda-Time or "defaultTimeZone" for JDK!

By default the `TimeZone` will be set to the JVM default, unless you produce a String annotated with `@DefaultTimeZone`. This can be achieved through either the Seam Config module or any bean that `@Produces` a method or field that matches the type and qualifier.

This will set the application time zone to be Tijuana:

```
@Produces
@DefaultTimeZone
private String defaultTimeZoneId = "America/Tijuana";
```

2.3. User TimeZone

In addition to the Application time zone, there is also a time zone assigned to each User Session.

```
@Inject
@Client
private DateTimeZone userDateTimeZone;

@Inject
@Client
private TimeZone userTimeZone;
```

It can also be accessed through EL using "userDateTimeZone" for Joda-Time and "userTimeZone" for JDK.

By default the `DateTimeZone` and `TimeZone` for a User Session is initialized to the same as the Application. However, changing the User's `DateTimeZone` and `TimeZone` is a simple matter of firing an event to update it. An example would be

```
@Inject
@Client
@Alter
private Event<DateTimeZone> dtzEvent;

@Inject
@Client
@Alter
private Event<TimeZone> tzEvent;

public void setUserDateTimeZone() {
   DateTimeZone dtzTijuana = DateTimeZone.forID("America/Tijuana");
   dtzEvent.fire(dtzTijuana);

   TimeZone tzTijuana = TimeZone.getTimeZone("America/Tijuana");
   tzEvent.fire(tzTijuana);
}
```

2.4. Available TimeZones

We've also provided a list of available TimeZones that can be accessed via

```
@Inject
private List<ForwardingDateTimeZone> dateTimeZones;

@Inject
private List<ForwardingTimeZone> timeZones;
```

Locales

3.1. Application Locale

In a similar fashion to TimeZones we have an Application `Locale`:

```
@Inject
private java.util.Locale lc;
```

accessible via EL with "defaultLocale".

By default the `Locale` will be set to the JVM default, unless you produce a String annotated with `@DefaultLocale`. This can be achieved through either the Seam Config module, with any bean that `@Produces` a method or field that matches the type and qualifier.

This will set the application language to be English with the country of US:

```
@Produces
@DefaultLocale
private String defaultLocaleKey = "en_US";
```

As you can see from the previous example, you can define the `Locale` with `lang_country_variant`. It's important to note that the first two parts of the locale definition are not expected to be greater than 2 characters otherwise an error will be produced and it will default to the JVM `Locale`.

3.2. User Locale

The Locale associated with the User Session can be retrieved by:

```
@Inject
@Client
private java.util.Locale locale;
```

which is EL accessible via `userLocale`.

By default the `Locale` will be that of the Application when the User Session is initialized. However, changing the User's `Locale` is a simple matter of firing an event to update it. An example would be:

```
@Inject
```

```
@Client
@Alter
private Event<java.util.Locale> localeEvent;

public void setUserLocale() {
   Locale canada = Locale.CANADA;
   localeEvent.fire(canada);
}
```

3.3. Available Locales

We've also provided a list of available Locales that can be accessed via:

```
@Inject
private List<java.util.Locale> locales;
```

The locales that will be returned as available can be defined by extending `LocaleConfiguration`. As seen here:

```
public class CustomLocaleConfiguration extends LocaleConfiguration {
   @PostConstruct
   public void setup() {
      addSupportedLocaleKey("en");
      addSupportedLocaleKey("fr");
   }
}
```

Messages

4.1. Message Creation

There are currently two ways to create a message within the module.

The first would mostly be used when you don't want to add the generated message directly to the UI, but want to log it out, or store it somewhere else

```
@Inject
private MessageFactory factory;

public String getMessage() {
    MessageBuilder builder = factory.info("There are {0} cars, and they are all {1}; {1} is the best color.", 5, "green");
    return builder.build().getText();
}
```

The second is to add the message to a list that will be returned to the UI for display.

```
@Inject
private Messages messages;

public void setMessage() {
    messages.info("There are {0} cars, and they are all {1}; {1} is the best color.", 5, "green");
}
```

Either of these methods supports the four message levels which are info, warning, error and fatal.

Both MessageFactory and Messages support four ways in which to create a Message:

- Directly adding the message
- Directly adding the message and replacing parameters
- Retrieving the message from a bundle
- Retrieving the message from a bundle and replacing parameters

Examples for each of these are:

```
messages.info("Simple Text");
```

```
messages.info("Simple Text with {0} parameter", 1);
```

```
messages.info(new BundleKey("org.jboss.international.seam.test.TestBundle", "key1"));
```

```
messages.info(new BundleKey("org.jboss.international.seam.test.TestBundle", "key2"), 1);
```

4.2. Properties Files

The examples in the previous section on how to create a message from a properties file made the assumption that you had already created it! Now we tell you how to actually do that.

When creating a `BundleKey` in the previous section, we were passing it a bundle name of "org.jboss.international.seam.test.TestBundle". This name is essentially the path to the properties file! Let me explain. As we all know properties files need to be on the classpath for our code to find them, so "org.jboss.international.seam.test.TestBundle" is telling our code that on the classpath there is a `TestBundle.properties` file located at a path of `org/jboss/international/seam/test`.

To create a property file for another language, it's simply a case of appending the name of the locale to the end of the file name. Such as `TestBundle_fr.properties` for French or `TestBundle_en_US.properties` for American English.

Note

If you only ever intend to use a single language within your application, there is no need to create a locale specific properties file, as the non locale version will be used if a locale specific properties file is not present.