Krail Documentation

Release 0.16.0.0

David Sowerby

Sep 08, 2018

Tutorial

This documentation is in 3 parts

- *Tutorial.* As you would expect, this is a step by step tutorial.
- *User Guide.* Notes on how to do things, collated by feature
- *Developer Guide.* Notes on why things are done the way they are. In truth, this is really a loose collection of technical notes optimistically called a "developer guide"

Please do give feedback, whether it is something you do like or something you do not like.

Enjoy

David Sowerby

Contents:

CHAPTER 1

Introduction to the Tutorial

Welcome to the Krail Tutorial. It is definitely better to work through the Tutorial in sequence, but if you want to skip a step, you will find a link at the end of each chapter so that you can clone each stage from a GitHub repository.

There is also a developers guide, but to be honest that is more like a collection of rough notes at the moment, until I can find time to write it properly.

Note: This tutorial is based on the Vaadin 7 version of Krail - it should still be possible to follow it, but it will not be updated until the move to Eclipse Vert.x is done (or fails!)

Please do give feedback, whether it is something you do like or something you do not like.

Enjoy

David Sowerby

CHAPTER 2

Getting Started

This Tutorial will take you through some of the basic steps of getting an application up and running, quickly, with Krail.

Krail encourages prototyping, by providing a lot of default functionality so that you can see early results. Most of that functionality can then be modified or replaced. The aim is to give the Krail developer the best of both worlds - quick results, but still the freedom to modify things however they wish.

2.1 Creating a Krail application with Gradle

2.1.1 Preparation

This tutorial assumes that you have Gradle already installed. The Vaadin Gradle plugin used here requires Gradle 4+

It is also assumed that you will be using Git for version control, and have it installed.

Earlier versions of this tutorial code allowed a download at every step, but regrettably that proved time consuming to maintain - there are some useful checkpoints in the code, though, if you look through the Git log

You can, however, download the code for the entire Tutorial from GitHub

2.1.2 Create a build file

Command Line

Change to your Git root directory, for example:

```
cd /home/david/git
```

Create a directory for your project (called "**krail-tutorial**" in this case), and initialise it for git.

```
mkdir krail-tutorial
cd krail-tutorial
git init
gedit build.gradle
```

You will now have an empty build file open. Cut and paste the following into the file & save it

```
 plugins {
 id "com.devsoap.plugin.vaadin" version "1.2.3"
 id 'eclipse-wtp'
 id 'idea'
 }

vaadin {
    version = '8.2.0' // This version should match that used by the version of Krail␣
↪you are using
    logToConsole = true
}

vaadinCompile {
    widgetset 'com.example.tutorial.widgetset.tutorialWidgetset'
}

sourceCompatibility = '1.8'

repositories {
    jcenter()
}

dependencies {
    compile(group: 'uk.q3c.krail', name: 'krail', version: '0.14.0.0')
}

configurations.all {
    resolutionStrategy {
        // GWT requires an old version of the validation API.  Changing to a newer␣
↪version breaks widgetset compile but throws no errors
        force 'javax.validation:validation-api:1.0.0.GA'
    }
}

task wrapper(type: Wrapper) {
    gradleVersion = '4.4.1'
}
```

- The first entry is for a Vaadin Gradle plugin, and provides some valuable Vaadin specific Gradle tasks
- The 'eclipse' and 'idea' plugins are optional, but useful for generating IDE specific files.
- We need to tell the vaadin-gradle plugin which version of Vaadin to use - this must match the version being used by the version of Krail selected
- Krail requires Java 8, hence the line "sourceCompatibility = '1.8'"
- Of course, you cannot do without Krail . . .
- There is a version conflict to resolve, between the dependencies of the various component parts of Krail. The ResolutionStrategy is there to resolve those version conflicts. GWT requires an older version of the javax validation API - if you don't force the correct version to be used, then the widgetset compile will fail - and

worse, it fails without any error messages.

- finally, the gradle-vaadin plugin needs Gradle at version 4+, so we will create a Gradle wrapper task

Now save the file and add it to Git

```
git add build.gradle
```

And finally, create a Gradle wrapper:

```
gradle wrapper
```

2.1.3 Create the Project

The Vaadin Gradle plugin makes things easier for us. From the command line:

```
gradle vaadinCreateProject
```

2.1.4 Import the Project to your IDE

IDEA

- From the command line

```
gradle idea
```

- In IDEA, start the import: File | Open and select *krail-tutorial/build.gradle*
  - In the import dialog:
    - Ensure that JDK 1.8 is selected
    - Use "default gradle wrapper"
    - Select "Create directories for empty content roots automatically"

    Tip

    IDEA may prompt you to add the project VCS root - say yes if it does.
- Delete the pacakge *com.example.krailtutorial* completely - we will create our own shortly. There are a couple of generated files in these folders, but they can be deleted.
- To reduce what goes in to Git, let's just add a simple .gitignore file at the project root:

```
.classpath
.idea
.project
build/*
out
classes
.gradle/

*.iml
*.ipr
*.iws
```

- Right click on the project folder and select Git | Add to add all files to Git.

2.1.5 Eclipse

Please see this contribution

2.1.6 Krail preparation

Guice and DI

This tutorial does not attempt to describe Guice, or Dependency Injection - which is what Krail is based on - but even if you are not familiar with either you may find that Krail is a good way to become so. The Guice documentation is a very good introduction to the principles - and for reference, Krail uses constructor injection with one or two specific exceptions.

Setting up the application

Let's keep all the application configuration in one place and create a package under src/main/java:

>com.example.tutorial.app

Create a Servlet

You may have noticed when you deleted the groovy folders, that a `TutorialServlet` had been generated. We do need one, but not that one!

In the com.example.tutorial.app package, create a class `TutorialServlet`, extended from `BaseServlet`:

```
package com.example.tutorial.app;

import com.google.inject.Inject;
import com.google.inject.Singleton;
import uk.q3c.krail.core.guice.BaseServlet;
import uk.q3c.krail.core.ui.ScopedUIProvider;

@Singleton
public class TutorialServlet extends BaseServlet {


@Inject
public TutorialServlet(ScopedUIProvider uiProvider) {
    super(uiProvider);
}
```

Define a Widgetset

If you are familiar with Vaadin, you will be familiar with widgetsets. However, if you are not, they can seem a bit of a mystery. The Vaadin documentation is generally very good, but one thing which does not seem to be clear is when to use the in-built widgetset and when to specify your own. We find it easier just to start by defining your own at the project set up stage. To set this up, we need to modify the Servlet:

```
@Singleton
public class TutorialServlet extends BaseServlet {

```

(continues on next page)

(continued from previous page)

```
    @Inject
    public TutorialServlet(ScopedUIProvider uiProvider) {
        super(uiProvider);
    }

     @Override
    protected String widgetset() {
        return "com.example.tutorial.widgetset.tutorialWidgetset";
    }
}
```

In the *build.gradle* file, add a vaadin closure set logToConsole - it provides a little extra console output during a build. It is useful, but not essential.

```
vaadin{
    logToConsole = true
    version = '7.7.10'
}
```

Complete Build file

The full *build.gradle* file should look like this:

```
plugins {
    id "com.devsoap.plugin.vaadin" version "1.2.3"
    id 'eclipse-wtp'
    id 'idea'
}

vaadin {
    version = '7.7.10' // This version should match that used by the version of Krail␣
↪you are using
    logToConsole = true
}

sourceCompatibility = '1.8'

repositories {
    jcenter()
}

dependencies {
    compile 'uk.q3c.krail:krail:0.10.0.0'
}

configurations.all {
    resolutionStrategy {
        // GWT requires an old version of the validation API.  Changing to a newer␣
↪version breaks widgetset compile but throws no errors
        force 'javax.validation:validation-api:1.0.0.GA'
    }
}

task wrapper(type: Wrapper) {
```

(continues on next page)

(continued from previous page)

```
    gradleVersion = '4.1'
}
```

Create a Servlet Module

In the *com.example.tutorial.app package*, create a class `TutorialServletModule`, extended from `BaseServletModule`:

```
package com.example.tutorial.app;

import uk.q3c.krail.core.guice.BaseServletModule;

public class TutorialServletModule extends BaseServletModule {

@Override
 protected void configureServlets() {
 serve("/*").with(TutorialServlet.class);
 }
}
```

Create a Binding Manager

In Krail terminology, the Binding Manager is a central point of Guice configuration. Guice modules specify how things are bound together, and the Binding Manager selects which modules to use. All Krail applications use their own Binding Manager, usually sub-classed from `DefaultBindingManager`. To create one for the tutorial:

In the com.example.tutorial.app package, create a class `BindingManager`, extended from `DefaultBindingManager`

```
package com.example.tutorial.app;

import com.google.inject.Module;
import uk.q3c.krail.core.guice.DefaultServletContextListener;

import java.util.List;

public class BindingManager extends DefaultBindingManager {

@Override
protected void addAppModules(List<Module> baseModules) {

}

@Override
protected Module servletModule() {
    return new TutorialServletModule();
}
```

Notice that we override `servletModule()` to let Guice know about our `TutorialServletModule`

Create web.xml

- Create a new directory, src/main/webapp/WEB-INF
- Then create a *web.xml* file. Note that the listener refers to our `BindingManager`. This could be the only xml you will use for the entire project

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.
↪com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/
↪xml/ns/javaee/web-app_3_0.xsd"
         id="WebApp_ID" version="3.0">
    <display-name>Krail Tutorial</display-name>
    <context-param>
        <description>
            Vaadin production mode
        </description>
        <param-name>productionMode</param-name>
        <param-value>false</param-value>
    </context-param>

    <filter>
        <filter-name>guiceFilter</filter-name>
        <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
        <async-supported>true</async-supported>
    </filter>
    <filter-mapping>
        <filter-name>guiceFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <listener>
        <listener-class>com.example.tutorial.app.BindingManager</listener-class>
    </listener>

</web-app>
```

Adding Some Pages

That's all the plumbing that is needed to get started - but we do not have any pages yet, so there's nothing to see. We will take a shortcut for the Tutorial and use some that already exists - you will see how the relationship between Guice modules and pages could be very convenient for building modular applications.

The `SystemAccountManagementPages` class in Krail is a set of not very useful pages (it just meant as an example) composed as a Guice module. We will add that module to the Binding Manager. Note that we use the `addSitemapModules()` method - we could just add all modules in `addAppModules()`, the separation is purely for clarity.

```
@Override
   protected void addSitemapModules(List<Module> baseModules) {
   baseModules.add(new SystemAccountManagementPages());
}
```

The complete `BindingManager` now looks like:

```
package com.example.tutorial.app;

import com.google.inject.Module;
import uk.q3c.krail.core.guice.DefaultServletContextListener;
import uk.q3c.krail.core.navigate.sitemap.SystemAccountManagementPages;

import java.util.List;

public class BindingManager extends DefaultBindingManager {

@Override
protected void addAppModules(List<Module> baseModules) {

}

@Override
protected Module servletModule() {
    return new TutorialServletModule();
}

@Override
protected void addSitemapModules(List<Module> baseModules) {
    baseModules.add(new SystemAccountManagementPages());
}
```

Theme(s)

You could actually launch the Tutorial application now, but if you did it would look terrible - it has no CSS applied. To give the application some style we need to apply a Vaadin theme. It is possible to use themes from the Vaadin theme jar, but it is advisable to extract them and serve them statically, as recommended by the Vaadin documentation:

The built-in themes included in the Vaadin library JAR are served dynamically from the JAR by the servlet. Serving themes and widget sets statically by the web server is more efficient._

So let's do that now.

- Find the vaadin-themes.jar. The easiest way is to search the {$user.home}/gradle directory - it should have been downloaded with the other Vaadin jars. If for any reason it is not there, you can download it from JCenter or Maven Central
- extract the jar
- locate the theme folders - you will find them in the VAADIN/themes folder
- copy folders for the themes you want - for the Tutorial, just copy all of them - into src/main/webapp/VAADIN/themes.
- delete the automatically created KrailTutorial theme

For readers less familiar with Vaadin, "reindeer" is the default style, and "valo" is the most recent.

Build and Run

The one aspect of the build that tends to give problems is the widgetset compile - it seems very sensitive. We therefore suggest compiling it first by executing:

```
gradle vaadinCompile
```

from either the command line or IDE. You can see whether it has compiled by checking the src/main/webapp/VAADIN/widgetsets folder - it should have contents. (A compile failure usually creates a widgetsets folder, but leaves it empty)

We can now build and run the application - set up a run configuration in your IDE to take the war output and run it on Tomcat or whichever application server you are using:

Run Configuration in IDEA

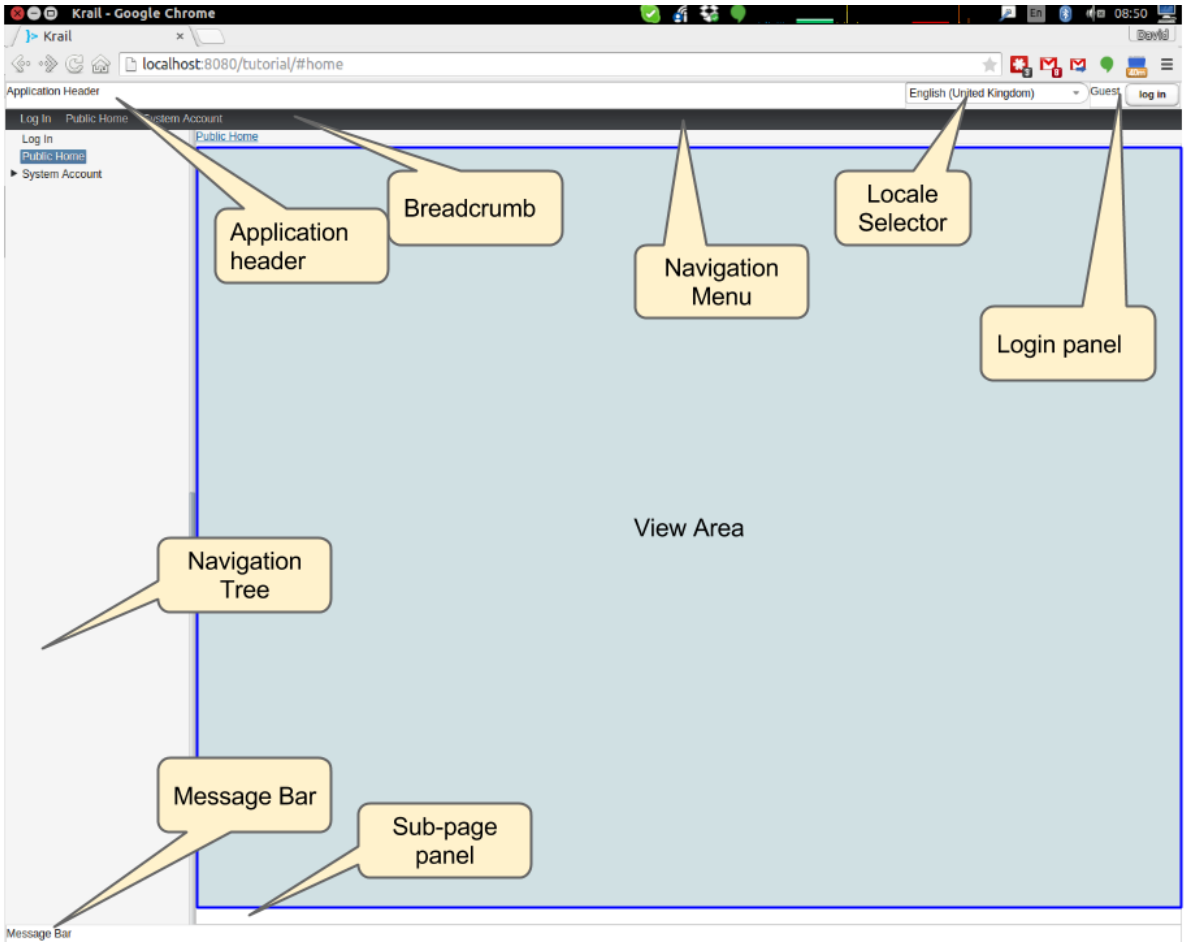
```
Run | Edit Configurations<br>
 + | Tomcat Server | Local<br>
Name: Tutorial<br>
Deployment: + | artifact | tutorial.war<br>
Application context: /tutorial
```

- refresh Gradle
- Build | Rebuild project
- Run Tutorial

Run Configuration in Eclipse

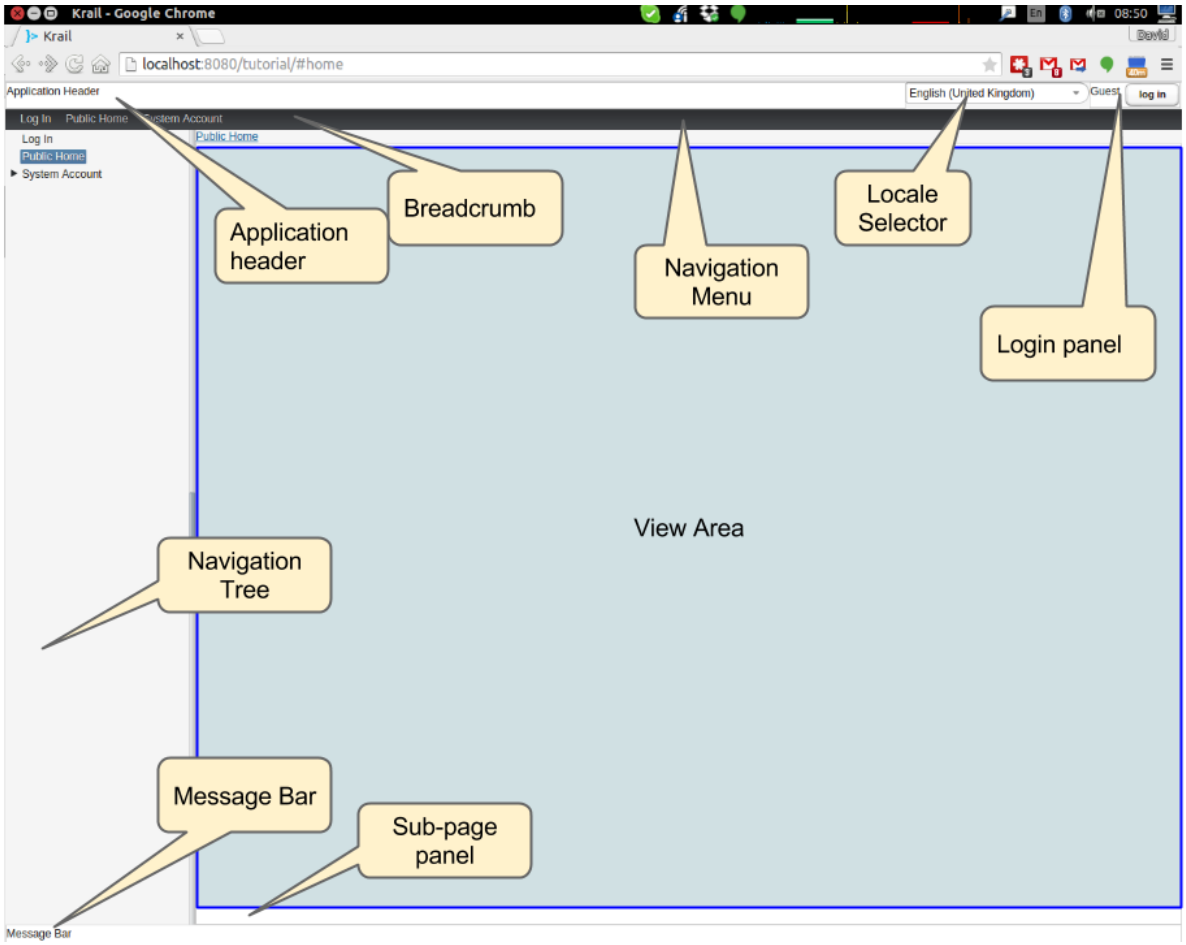
tbd

. . . . you should now see something like this:



2.2 Exploring the Basic Application

There are a few things to see, even in this very basic implementation.



- The "screen" presentation is provided by `DefaultApplicationUI` - UI in this context refers to the Vaadin concept of UI, which is generally equivalent to a browser tab.
- `DefaultApplicationUI` contains a number of components and both the UI and its components can be replaced, sub-classed or modified. All the parts described below are pluggable components, held by the UI:
  - The Application Header is just a panel to hold things like logos
  - The navigation tree, navigation menu, breadcrumb and sub-page panel menu are all navigation-aware components. You can navigate pages by clicking on any of them, or just change the URL directly. These navigation components are tied to a Sitemap, which defines the structure of the site, and the Views used to represent each page. You will see how this works when we create some new pages.
  - The Locale selector will not do much yet, as there are no alternative Locales defined - that will be covered later in the Tutorial.
  - The login panel offers a login button and a login status - we will log in in a moment
  - The message bar is just a place for messages to the user.
  - The View area (in blue) is where all the work is done - it is here that you will put forms and tables etc.

Of course, as a developer, you will almost certainly have logged in by now, but in case you have not - you can use any user name, and a password of "password", so that you can pretend to be a real user with a memorable password ...

A couple of things have changed now you have logged in:

- You will no longer be on the login page - that's a bit obvious, but it is worth noting that even the rules for where to navigate to after log in is a replaceable element.

- There is now an extra page in the navigation components, called 'Private' - this represents a restricted area of the site, where only authorised users can have access. The other pages are all "public".
- The login panel shows your user name, and now offers a "logout" button.

This is achieved using two major components, the `DefaultRealm` (a very simple implementation of the Apache Shiro Realm) and `PageController`, a Krail component used to control the display of pages authorised by your Realm implementation. We will come back to these when we look at User Access Control.

Now try this sequence:

- Login
- Click on "Private" and you will see that it jumps to "Private home" - this is configurable behaviour - it is a redirect so that there does not need to be a view if the "Private" page itself will never be shown
- Logout. You will now be on the logout page (which by default does not appear in the navigation components - also configurable behaviour)
- Press the browser 'back' button - and a notification will pop up saying that *"private/home is not a valid page"*. Even though you know this is not the case, this message is deliberate, as it means that if a user tries to guess a url that they are not authorised for, they will not even get confirmation that the page exists.
- Look at the message bar and you will see that the same message has appeared there. We will look at user notifications and how they are handled a bit later.

It should be noted that although the Tutorial uses the idea of a 'private' set of pages, how you define and authorise access to pages is extremely flexible, and mostly a matter of how you want to do it.

2.3 Summary

You have created a basic application, and can have already seen:

- Integration with User Access Control from Apache Shiro
- a pluggable set of pages
- Navigation aware components acting together
- User notifications

2.4 Download from GitHub

To get to this point straight from GitHub:

```
git clone https://github.com/davidsowerby/krail-tutorial.git
cd krail-tutorial
git checkout --track origin/krail_0.10.0.0
```

Revert to commit *Getting Started completed*

CHAPTER 3

Page Navigation

Clearly we will want to add some new pages, but first we must know what constitutes the definition of a page

3.1 Defining a Page

A page is represented by a URI, which maps to a specified `KrailView` class. The name of the page is presented to the user in navigation aware components, so that name must be Locale sensitive. Once the page is defined, it becomes part of the Krail `Sitemap`, which forms the heart of the navigation system.

There are two ways to add pages to Krail and make use of the navigation features, and you can use either one, or both. These are the "direct" method or "annotation" method. We will use both methods.

Because the page name is locale sensitive, we will first need to provide I18N support.

3.2 Introducing I18N

You may think that it is premature to be considering I18N at this stage - especially if you are writing an application which will only use one language. However, Krail treats I18N as a first class citizen, and you will find the result of these steps surprisingly useful even in a single Locale application. You could read the full I18N description now, or just follow these steps, as we will come back to I18N later in the Tutorial.

3.2.1 Create an I18N Annotation

- create a package 'i18n', under 'com.example.tutorial'
- create two Enum classes, one called 'LabelKey' and one called 'DescriptionKey'. Each should implement the `I18NKey` interface

```
package com.example.tutorial.i18n;
import uk.q3c.krail.i18n.I18NKey;

public enum LabelKey implements I18NKey {
}
```

```
package com.example.tutorial.i18n;
import uk.q3c.krail.i18n.I18NKey;

public enum DescriptionKey implements I18NKey {
}
```

The names of these classes can be anything, it is the `I18NKey` interface which is important.

This is all we need for our I18N integration for now, so we can get on with adding pages.

3.3 Add a Page - direct method

The "direct" method simply means pages are defined directly in a Guice module. We will start by adding some private pages ("private" means they will be available only to authorised users).

- To keep our pages separate, create a package 'pages', under 'com.example.tutorial'
- Create a class 'MyPages' and extend it from `DirectSitemapModule` and provide
- implement the abstract `define()` method

```
package com.example.tutorial.pages;

import uk.q3c.krail.core.navigate.sitemap.DirectSitemapModule;

public class MyPages extends DirectSitemapModule{

    @Override
    protected void define() {

    }
}
```

We will use the `define()` method to provide our page definitions. We will create three pages, one at the the site root, with two sub-pages, which we want to look something like this in the navigation tree:

```
> Finance
>> Accounts
>> Payroll
```

- enter the following in the `define()` method

```
package com.example.tutorial.pages;

import uk.q3c.krail.core.navigate.sitemap.DirectSitemapModule;
import uk.q3c.krail.core.shiro.PageAccessControl;
import com.example.tutorial.i18n.LabelKey;

public class MyPages extends DirectSitemapModule{

```

(continues on next page)

(continued from previous page)

```
    @Override
    protected void define() {
        addEntry("private/finance", FinanceView.class, LabelKey.Finance,
                PageAccessControl.PERMISSION);
        addEntry("private/finance/accounts", AccountsView.class, LabelKey.Accounts,
                PageAccessControl.PERMISSION);
        addEntry("private/finance/payroll", PayrollView.class, LabelKey.Payroll,
                PageAccessControl.PERMISSION);
    }
}
```

Make sure you get the right LabelKey - there is one in Krail core as well.

You will have compile errors, but let's look at what these entries mean.

- The first parameter is the URI segment, and we generally keep to all lowercase. The second and third entries are subpages, so need a qualified path.
- The second parameter is the class to use as a View - we haven't created them yet.
- The third parameter is the page name, is locale-sensitive and therefore an `I18NKey`
- The fourth parameter determines what sort of access control is applied to the page. We want controlled access to these pages, so this parameter is set to PERMISSION

We'll make it easier by extending the `Grid3x3ViewBase` base class from the Krail core - this just gives us a 3x3 grid to place components in.

Tip: Extending ViewBase or one of its sub-classes is usually the easiest way to create your views, but however you do it, you must implement `KrailView`. `ViewBase` can also help with deserialization.

- create the 3 views we want ... AccountsView, FinanceView and PayrollView ... just by extending `Grid3x3ViewBase` and injecting `Translate` (only FinanceView is shown here):

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

public class FinanceView extends Grid3x3ViewBase {

    @Inject
    protected FinanceView(Translate translate) {
        super(translate);
    }
}
```

3.3.1 Defining the I18NKeys

By default, if Krail's `I18NProcessor` cannot find the value of an `I18NKey`, it uses the name of the enum instead, with underscores replaced with spaces. This means that as long as you are comfortable with breaking the 'all-uppercase' convention for enum constant names, you can get started quickly by not defining any values for the I18NKeys. This is great for prototyping, and even if your application uses a language with accents and diacriticals, the enum name may be good enough for a prototype.

- Add the required constants to LabelKey

```
package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.I18NKey;

public enum LabelKey implements I18NKey {
    Accounts, Payroll, Finance
}
```

3.4 Using the Pages

Now we have defined the pages in a Guice module, we need to tell the `BindingManager` to include them:

```
@Override
   protected void addSitemapModules(List<Module> baseModules) {
       baseModules.add(new SystemAccountManagementPages());
       baseModules.add(new MyPages());
}
```

3.5 View the Pages

Run the application again. When the application starts the new pages will not be visible - but that is what we should expect, as we said these pages needed permission to view.

- Log in (any username, password='password'), and you will see the pages, under 'Private', in the navigation tree and menu.

You may be wondering whether these pages need to be under the 'Private' branch. At the moment they do, but only because of the very simple access control rules supplied by `DefaultRealm`. You can actually define any logical structure, and we will see how to control permissions in the User Access Control section of the Tutorial.

3.6 Add a Page - Annotation method

The second method of defining a page is to use an annotation on a `KrailView` implementation. To begin with, we need to tell Krail where to look for annotated views - this reduces the amount of scanning Krail has to do at start up. To do that we:

- create a new class in the 'pages' package, "AnnotatedPagesModule" and extend `AnnotationSitemapModule`
- implement the `define()` method
- add an entry in the define method, as below:

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.LabelKey;
import uk.q3c.krail.core.navigate.sitemap.AnnotationSitemapModule;

public class AnnotatedPagesModule extends AnnotationSitemapModule {

```

(continues on next page)

(continued from previous page)

```
    @Override
    protected void define() {
        addEntry("com.example.tutorial.pages",LabelKey.Accounts);
    }
}
```

The call to `addEntry` tells Krail to recursively scan the *com.example.tutorial.pages* package for classes with a `@View annotation`. Multiple calls to `addEntry` can be made. The second parameter should be an `I18NKey` from the same enum that you are going to use in your `@View` annotations. The value you supply to the `addEntry` method is just a sample, it just needs to be from the same class. This is necessary because of the limitations on what Java allows as `Annotation` parameter types

Now that this is done, any views in the 'pages' package, annotated with `@View`, will be added to the Sitemap.

- create another view, "PurchasingView" in the pages package, sub-classed from `Grid3x3ViewBase`:

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.navigate.sitemap.View;
import uk.q3c.krail.core.shiro.PageAccessControl;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

@View(uri = "private/finance/purchasing",pageAccessControl = PageAccessControl.
↪PERMISSION,labelKeyName = "Purchasing")
public class PurchasingView extends Grid3x3ViewBase {

    @Inject
    protected PurchasingView(Translate translate) {
        super(translate);
    }
}
```

- create the 'Purchasing' constant for `LabelKey` ' public enum LabelKey implements I18NKey { Accounts, Payroll, Finance, Purchasing } '
- tell the `BindingManager` to include the module we have just created

```
@Override
   protected void addSitemapModules(List<Module> baseModules) {
       baseModules.add(new SystemAccountManagementPages());
       baseModules.add(new MyPages());
       baseModules.add(new AnnotatedPagesModule());
   }
```

- Run the application, log in and you will see that "Purchasing" has been added to the Finance page.

3.7 Choosing the Method

You can mix Direct and Annotation sitemap entries however you wish, but that can get a bit confusing to manage. Which method you choose is mostly a matter of preference, but there is one feature of the direct method you should be aware of.

Our direct pages module looks currently looks like this: ' addEntry("private/finance", FinanceView.class, LabelKey.Finance, PageAccessControl.PERMISSION); addEntry("private/finance/accounts", AccountsView.class, La-

belKey.Accounts, PageAccessControl.PERMISSION); addEntry("private/finance/payroll", PayrollView.class, LabelKey.Payroll, PageAccessControl.PERMISSION); ' There is a lot of repetition in the URIs, so there is an alternative, by setting a `rootURI` which is applied to all pages:

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.LabelKey;
import uk.q3c.krail.core.navigate.sitemap.DirectSitemapModule;
import uk.q3c.krail.core.shiro.PageAccessControl;

public class MyPages extends DirectSitemapModule {

[source,java]
----
public MyPages() {
    rootURI = "private/finance";
}

@Override
protected void define() {
    addEntry("", FinanceView.class, LabelKey.Finance,
            PageAccessControl.PERMISSION);
    addEntry("accounts", AccountsView.class, LabelKey.Accounts,
            PageAccessControl.PERMISSION);
    addEntry("payroll", PayrollView.class, LabelKey.Payroll,
            PageAccessControl.PERMISSION);
}
----

}
```

- update MyPages so it is as above
- run the application and you will see that the pages appear in the same way as before

3.8 Moving a Set of Pages

We can easily move all the pages of a Direct module by changing the `rootUri` - they can be moved anywhere in the Sitemap, as a set, as long the Sitemap maintains a logical structure. We will need to keep the finance pages on the "Private" branch for now, because of the Access Control rules, but as an example, let's suppose we decide that it should have a rootURI of "private/finance-department" instead:

- modify the Binding Manager as below, to provide a different rootURI as the module is constructed:

```
@Override
  protected void addSitemapModules(List<Module> baseModules) {
      baseModules.add(new SystemAccountManagementPages());
      baseModules.add(new MyPages().rootURI("private/finance-department"));
      baseModules.add(new AnnotatedPagesModule());
  }
```

- modify the annotated view (otherwise the Sitemap will break because there is no longer a "private/finance" page

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
```

(continues on next page)

(continued from previous page)

```
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.navigate.sitemap.View;
import uk.q3c.krail.core.shiro.PageAccessControl;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

@View(uri = "private/finance-department/purchasing",pageAccessControl =␣
↪PageAccessControl.PERMISSION,labelKeyName = "Purchasing")
public class PurchasingView extends Grid3x3ViewBase {

    @Inject
    protected PurchasingView(Translate translate) {
        super(translate);
    }
}
```

- Run the application and check that new URI is being used.

Tip: If you do want to set the rootURI directly in the module, you need to do so in the constructor, or it will prevent the fluent method shown above from working.

Tip: This feature of moving blocks of pages is available only with Direct pages. Although it might be possible to do something similar with annotated pages by mapping packages to URIs, there are currently no plans to do so.

3.9 Navigation

3.9.1 Add some public pages

Add a couple of public pages:

- in the pages package create "MyPublicPages" class, extended from `DirectSitemapModule` with a couple of pages defined. Note that we are going to put these as 'roots' of the tree, as `rootUri` is set to an empty string.:

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.LabelKey;
import uk.q3c.krail.core.navigate.sitemap.DirectSitemapModule;
import uk.q3c.krail.core.shiro.PageAccessControl;

public class MyPublicPages extends DirectSitemapModule {


public MyPublicPages() {
    rootURI = "";
}

@Override
protected void define() {
    addEntry("news", NewsView.class, LabelKey.News, PageAccessControl.PUBLIC);
    addEntry("contact-us", ContactUsView.class, LabelKey.Contact_Us,␣
↪PageAccessControl.PUBLIC);
```

(continues on next page)

(continued from previous page)

```
}
```

- Create the views, extended from‘Grid3x3ViewBase‘:

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

public class ContactUsView extends Grid3x3ViewBase {

@Inject
protected ContactUsView(Translate translate) {
    super(translate);
}
```

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

public class NewsView extends Grid3x3ViewBase {


@Inject
protected NewsView(Translate translate) {
    super(translate);
}
```

- And add the `LabelKey` constants

```
public enum LabelKey implements I18NKey {
 Accounts, Payroll, Finance, News, Contact_Us, Purchasing
}
```

- Finally, update the `BindingManager` to include this new set of pages:

```
@Override
   protected void addSitemapModules(List<Module> baseModules) {
       baseModules.add(new SystemAccountManagementPages());
       baseModules.add(new MyPages().rootURI("private/finance-department"));
       baseModules.add(new AnnotatedPagesModule());
       baseModules.add(new MyPublicPages());
   }
```

3.9.2 Getting the Navigator

We will do just a little bit more with these views to help demonstrate navigation - we'll just add some buttons to direct us to different URIs. First, though, we need access to Krail's `Navigator`. We will inject it into both views, using constructor injection:

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.navigate.Navigator;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

public class ContactUsView extends Grid3x3ViewBase {

private Navigator navigator;

@Inject
protected ContactUsView(Translate translate, Navigator navigator) {
    super(translate);
    this.navigator = navigator;
}
```

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.navigate.Navigator;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

public class NewsView extends Grid3x3ViewBase {

private Navigator navigator;

@Inject
protected NewsView(Translate translate, Navigator navigator) {
    super(translate);
    this.navigator = navigator;
}
```

3.9.3 Adding some components

- Add buttons and actions in the `doBuild` method of `NewsView`:

```
@Override
 protected void doBuild(ViewChangeBusMessage busMessage) {
    super.doBuild(busMessage);
    Button navigateToContactUsBtn = new Button("Contact Us");
    Button navigateToPrivatePage = new Button("Accounts");
    navigateToContactUsBtn.addClickListener(c -&gt; navigator.navigateTo("contact-us
↪"));
    navigateToPrivatePage.addClickListener(c-&gt;navigator.navigateTo("private/
↪finance-department/accounts"));
    setCentreCell(new VerticalLayout(navigateToContactUsBtn,navigateToPrivatePage));
 }
```

The first two lines just create the buttons. The second two lines add click listeners, which are set up to use the‘Navigator‘to direct us to the chosen page. Then the buttons are added to a `VerticalLayout` which is put in the centre cell of the grid.

- Run the application, **but do not log in.**
- Click on the “News” page

- Press the "Contact Us" button, and you will be taken to the "Contact Us" page
- Press the browser back button, and you will be back on the "News" page
- Press the "Accounts" button - and you a notification will appear to say that the page does not exist. As mentioned earlier, the same notification is given whether you are not authorised or the page does not exist.
- Log in
- Press the "Accounts" button again, and as you are now authorised, you will be at the "Accounts" page

3.9.4 Navigating with Parameters

A common requirement is to land on a page with parameters - a record id, for example, so the page know which data to load. We are going to add a "Contact Detail" page to simulate this.

- Just as we've done before, add the page to ''MyPublicPages'', create the view and add the `LabelKey` constant:

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.LabelKey;
import uk.q3c.krail.core.navigate.sitemap.DirectSitemapModule;
import uk.q3c.krail.core.shiro.PageAccessControl;

public class MyPublicPages extends DirectSitemapModule {

public MyPublicPages() {
    rootURI = "";
}

@Override
protected void define() {
    addEntry("news", NewsView.class, LabelKey.News, PageAccessControl.PUBLIC);
    addEntry("contact-us", ContactUsView.class, LabelKey.Contact_Us,␣
↪PageAccessControl.PUBLIC);
    addEntry("contact-us/contact-detail", ContactDetailView.class, LabelKey.Contact_
↪Detail, PageAccessControl.PUBLIC);
}
```

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.Grid3x3ViewBase;

public class ContactDetailView extends Grid3x3ViewBase {

@Inject
protected ContactDetailView(Translate translate) {
    super(translate);
}
```

Receiving parameters

To set `ContactDetailView` up to receive parameters all we need to do is override either the `afterBuild` method or the `loadData` method. Using `loadData` (even if you are not loading data) means you won't forget to

call **super.afterBuild()** first . . .

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import com.vaadin.ui.FormLayout;
import com.vaadin.ui.Label;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.AfterViewChangeBusMessage;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;

public class ContactDetailView extends Grid3x3ViewBase {
 private Label idLabel;
 private Label nameLabel;

@Inject
protected ContactDetailView(Translate translate) {
    super(translate);
}

@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    super.doBuild(busMessage);
    idLabel = new Label();
    idLabel.setCaption("id");
    nameLabel = new Label();
    nameLabel.setCaption("name");
    setCentreCell(new FormLayout(idLabel, nameLabel));
}

@Override
protected void loadData(AfterViewChangeBusMessage busMessage) {
    idLabel.setValue(busMessage.getToState()
            .getParameterValue("id"));
    nameLabel.setValue(busMessage.getToState()
            .getParameterValue("name"));
}
```

The process in `loadData()` is straightforward. The busMessage is just an event, and it carries a reference to the navigation state we are navigating from, and the state we are navigating to. This is represented by `NavigationState`, which also contains any parameters that have been passed with the URI.

Sending parameters

To send parameters, construct a `NavigationState`, specifying the parameters to go with it and call `Navigator.navigateTo(NavigationState)`

- Update `ContactUsView` to add a button whose click listener builds the `NavigationState`, adds parameters, then calls the `Navigator`.

```
@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    super.doBuild(busMessage);
    Button navigateWithParametersBtn = new Button("Navigate with parameters");
    NavigationState navState = new NavigationState().virtualPage("contact-us/contact-
↪detail")
```

(continues on next page)

(continued from previous page)

```
                                                        .parameter("id", "33")
                                                    .parameter("name", "David");
    navigateWithParametersBtn.addClickListener(c->navigator.navigateTo(navState));
    setCentreCell(navigateWithParametersBtn);
}
```

- Run the application
- select "Contact Us"
- click on "Navigate with Parameters"
- You will now be at the "Contact Detail" page with the parameter values displayed.

3.9.5 Excluding a page from Navigation

If you think about the use of the "Contact Detail" page, it does not actually make sense for it to appear in the navigation components - the only time you would want to access this page is with some parameters to set its contents:

- Modify the page entry in `MyPublicPages`, by setting the positionIndex parameter to < 0

```
   @Override
protected void define() {
          addEntry("news", NewsView.class, LabelKey.News, PageAccessControl.PUBLIC);
          addEntry("contact-us", ContactUsView.class, LabelKey.Contact_Us,␣
↪PageAccessControl.PUBLIC);
          addEntry("contact-us/contact-detail", ContactDetailView.class, LabelKey.
↪Contact_Detail, PageAccessControl.PUBLIC,-1);
}
```

- Run the application, and the page will no longer appear in the navigation components, but is actually still there:
  - Go to the "Contact Us" page
  - Press the "Navigate with Parameters" button
  - The "Contact Detail" page appears as before.

3.10 Summary

- You have explored two methods of defining new pages, using Direct and Annotated methods.
- You have created navigation actions from code
- You have passed parameters to a page, as you typically might to load data
- You have excluded a page from navigation, but still make it part of the Sitemap
- You have "attached" an existing set of pages to a part of the Sitemap different from its default location

3.11 Download from GitHub

To get to this point straight from GitHub:

```
git clone https://github.com/davidsowerby/krail-tutorial.git
cd krail-tutorial
git checkout --track origin/krail_0.10.0.0
```

Revert to commit *Pages and Navigation Complete*

CHAPTER 4

Themes

The Vaadin handbook provides a full explanation of its architecture, and the role of the UI component.

For the purposes of this Tutorial, it is enough to consider the `UI` to be a representation of a browser tab. The `DefaultApplicationUI` is provided by Krail as a start point, but you may want to change elements of it or replace it completely. The first Tutorial section gave an overview of the DefaultApplicationUI if you need a refresher.

4.1 Replacing a UI

To use your own UI:

- in the package *com.example.tutorial.app*, create a class TutorialUI, and sub-class `DefaultApplicationUI`. As you can see, it uses a lot of injected objects - hopefully your IDE will create the constructor for you.
- don't forget the **@Inject** annotation for the constructor - it is very easy to miss when using IDE auto-completion

```
package com.example.tutorial.app;

import com.google.inject.Inject;
import com.vaadin.data.util.converter.ConverterFactory;
import com.vaadin.server.ErrorHandler;
import uk.q3c.krail.i18n.CurrentLocale;
import uk.q3c.krail.core.i18n.I18NProcessor;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.navigate.Navigator;
import uk.q3c.krail.option.Option;
import uk.q3c.krail.core.push.Broadcaster;
import uk.q3c.krail.core.push.PushMessageRouter;
import uk.q3c.krail.core.ui.ApplicationTitle;
import uk.q3c.krail.core.ui.DefaultApplicationUI;
import uk.q3c.krail.core.user.notify.VaadinNotification;
import uk.q3c.krail.core.view.component.*;

public class TutorialUI extends DefaultApplicationUI {
```

(continues on next page)

(continued from previous page)

```
[source]
----
@Inject
protected TutorialUI(Navigator navigator, ErrorHandler errorHandler, ConverterFactory␣
↪converterFactory, ApplicationLogo logo, ApplicationHeader header, UserStatusPanel␣
↪userStatusPanel, UserNavigationMenu menu, UserNavigationTree navTree, Breadcrumb␣
↪breadcrumb, SubPagePanel subpage, MessageBar messageBar, Broadcaster broadcaster,␣
↪PushMessageRouter pushMessageRouter, ApplicationTitle applicationTitle, Translate␣
↪translate, CurrentLocale currentLocale, I18NProcessor translator, LocaleSelector␣
↪localeSelector, VaadinNotification vaadinNotification, Option option) {
    super(navigator, errorHandler, converterFactory, logo, header, userStatusPanel,␣
↪menu, navTree, breadcrumb, subpage, messageBar, broadcaster,
            pushMessageRouter, applicationTitle, translate, currentLocale, translator,
↪ localeSelector, vaadinNotification, option);
}
----

}
`
- Configure the`BindingManager`to use this new UI.  In this case we override a␣
↪specific`BindingManager`method, because we need to override the default:
`
@Override
protected Module uiModule() {
 return new DefaultUIModule().uiClass(TutorialUI.class).applicationTitleKey(LabelKey.
↪Krail_Tutorial);
}
`
- Add the new key to`LabelKey```, which your IDE will probably do for you.
- Run the application and confirm that the application title has changed in the␣
↪browser tab

= Themes

At the moment there is no alternative for setting the theme except by using the␣
↪*@Theme* annotation provided by Vaadin. On the new `TutorialUI`
- Set the theme with @Theme("valo")
```

@Theme(“valo”) public class TutorialUI extends DefaultApplicationUI {

```
@Inject
protected TutorialUI(Navigator navigator, ErrorHandler errorHandler, ConverterFactory␣
↪converterFactory, ApplicationLogo logo, ApplicationHeader header, UserStatusPanel␣
↪userStatusPanel, UserNavigationMenu menu, UserNavigationTree navTree, Breadcrumb␣
↪breadcrumb, SubPagePanel subpage, MessageBar messageBar, Broadcaster broadcaster,␣
↪PushMessageRouter pushMessageRouter, ApplicationTitle applicationTitle, Translate␣
↪translate, CurrentLocale currentLocale, I18NProcessor translator, LocaleSelector␣
↪localeSelector, VaadinNotification vaadinNotification, Option option) {
    super(navigator, errorHandler, converterFactory, logo, header, userStatusPanel,␣
↪menu, navTree, breadcrumb, subpage, messageBar, broadcaster,
            pushMessageRouter, applicationTitle, translate, currentLocale, translator,
↪ localeSelector, vaadinNotification, option);

}
```

}

```
- Run the application and observe the different appearance.

Valo is the most recent theme from Vaadin.  "Reindeer" is the default, which you have␣
↪been using until now.  For more information about themes, see the [Vaadin␣
↪Documentation](https://vaadin.com/book/-/page/themes.html).

#Summary
This was a short tutorial, covering the creation of a new UI, registering it, and␣
↪setting a Theme.

#Download from GitHub
To get to this point straight from GitHub:
```

git clone https://github.com/davidsowerby/krail-tutorial.git cd krail-tutorial git checkout –track origin/krail_0.10.0.0

```
Revert to commit _UI &amp;amp; Theme complete_
```

CHAPTER 5

User Notifications

Notifying users with messages seems a small topic, and typical UI code contains numerous calls to message boxes of one form or another. Consistency, however, can easily be lost, especially when there is a need for I18N as well. There are also times when you want the message to go to more than one place - for example both a splash message, and repeated in the message bar at the bottom of the screen as you have already seen.

Vaadin provides the `Notification` specifically for that purpose.

Krail provides a mechanism to support any method of presenting the message, but the message despatch is always from the `UserNotifier`

5.1 Sending the Message

- Make the `UserNotifier` available to the `NewsView` by injecting it into the constructor
- Add the *sendNotificationBtn* button
- Set the button's click listener to despatch the notification with a call to `userNotifier.notifyError`. There are warning and information calls available as well.
- add the button to the `VerticalLayout` in the call to `setCentreCell`

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.LabelKey;
import com.google.inject.Inject;
import com.vaadin.ui.Button;
import com.vaadin.ui.VerticalLayout;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.navigate.Navigator;
import uk.q3c.krail.core.user.notify.UserNotifier;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
```

(continues on next page)

(continued from previous page)

```
public class NewsView extends Grid3x3ViewBase {

    private Navigator navigator;
    private UserNotifier userNotifier;

    @Inject
    protected NewsView(Translate translate,Navigator navigator, UserNotifier␣
↪userNotifier) {
        super(translate);
        this.navigator = navigator;
        this.userNotifier = userNotifier;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        Button navigateToContactUsBtn = new Button("Contact Us");
        Button navigateToPrivatePage = new Button("Accounts");
        Button sendNotificationBtn = new Button("Send notification");
        navigateToContactUsBtn.addClickListener(c -> navigator.navigateTo("contact-us
↪"));
        navigateToPrivatePage.addClickListener(c -> navigator.navigateTo("private/
↪finance-department/accounts"));
        sendNotificationBtn.addClickListener((c -> userNotifier.notifyError(LabelKey.
↪Do_Not_do_That)));
        setCentreCell(new VerticalLayout(navigateToContactUsBtn,␣
↪navigateToPrivatePage, sendNotificationBtn));
    }
}
```

5.2 Current methods of presentation

If you look at the constructor for `TutorialUI` you will see the `MessageBar` and `VaadinNotification` instances being injected. The `MessageBar` is the component presented at the bottom of the screen, and `VaadinNotification` is a wrapper for the Vaadin Notification class. Both just listen for notification messages via the Event Bus ‘ @Inject protected TutorialUI(Navigator navigator, ErrorHandler errorHandler, ConverterFactory converterFactory, ApplicationLogo logo, ApplicationHeader header, UserStatusPanel userStatusPanel, UserNavigationMenu menu, UserNavigationTree navTree, Breadcrumb breadcrumb, SubPagePanel subpage, MessageBar messageBar, Broadcaster broadcaster, PushMessageRouter pushMessageRouter, ApplicationTitle applicationTitle, Translate translate, CurrentLocale currentLocale, I18NProcessor translator, LocaleSelector localeSelector, VaadinNotification vaadinNotification, Option option) { ‘ - Run the application and go to the “News Page”, press the “Send Notification” button, and the message will appear as a Vaadin ‘Splash’ notification and in the message bar at the bottom of the screen.

5.3 Different methods of presentation

If you wanted to provide your own methods of presenting user notifications, it is very easy to do, while still keeping the consistency of a single despatch point for user notifications - just copy the structure of `DefaultVaadinNotification` and provide your own method of presenting the messages.

5.4 Summary

At first this seems an almost trivial topic, but we would strongly recommend using `UserNotifier` from the start. This will give you consistency, and enable a very quick and simple change of notification method(s) later.

5.5 Download from GitHub

To get to this point straight from GitHub:

```
git clone https://github.com/davidsowerby/krail-tutorial.git
cd krail-tutorial
git checkout --track origin/krail_0.10.0.0
```

Revert to commit *User Notification Complete*

CHAPTER 6

Options

Krail sees Options as the top layer of configuration. Options give users as much control as the Krail developer wants to give them, at runtime. They can be used for anything which you would typically find in a settings / preferences / options menu.

6.1 Out of the Box

Let's start with what Krail provides out of the box, the `SimpleHierarchy`. When looking for option values, this provides 3 levels:

- the user level value
- the system level value
- a default, hard-coded value

The process is very simple - starting from the top of the hierarchy, the user level, Krail looks for the first defined value, and uses that. The user and system level would normally be in persistence, and the default coded level is there so that even if persistence is inaccessible, or not yet populated, the system behaves in a predictable way.

This could be described as the user level value overriding the system level, which in turn overrides the default coded level.

6.2 Working example

We will demonstrate this with a page on which the user can select the news topics they wish to see.

- In the 'pages' package create a new view, 'MyNews', extended from `Grid3x3View`
- Add 3 Labels with some example text, for CEO News, Items for Sale and Vacancies

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import com.vaadin.ui.Label;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;

public class MyNews extends Grid3x3ViewBase {

    @Inject
    protected MyNews(Translate translate) {
        super(translate);
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        Label ceoNews = new Label("CEO News");
        Label itemsForSale = new Label("Items for Sale");
        Label vacancies = new Label("Vacancies");
        ceoNews.setSizeFull();
        itemsForSale.setSizeFull();
        vacancies.setSizeFull();
        setMiddleLeft(itemsForSale);
        setCentreCell(ceoNews);
        setMiddleRight(vacancies);
    }
}
```

- In the 'pages' package create a new direct pages module, "MyOtherPages"

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.LabelKey;
import uk.q3c.krail.core.navigate.sitemap.DirectSitemapModule;
import uk.q3c.krail.core.shiro.PageAccessControl;

public class MyOtherPages extends DirectSitemapModule {
 /**
 * {@inheritDoc}
 */
 @Override
 protected void define() {
 addEntry("private/my-news", MyNews.class, LabelKey.My_News, PageAccessControl.
↪PERMISSION);
 }
}
`
- Add this new module to the`BindingManager`
`
 @Override
 protected void addSitemapModules(List<Module> baseModules) {
 baseModules.add(new SystemAccountManagementPages());
 baseModules.add(new MyPages().rootURI("private/finance-department"));
 baseModules.add(new AnnotatedPagesModule());
 baseModules.add(new SystemAdminPages());
```

(continues on next page)

(continued from previous page)

```
baseModules.add(new MyPublicPages());
baseModules.add(new MyOtherPages());
}
```

- add the constant "My_News" to `LabelKey`
- run the application, log in and navigate to "My News" just to make sure it works. You should see the three items across the centre of the page.

At the moment these "news channels" will always appear. Now we need to make them optional - after all, you may not want to see the vacancies, but you will always want to see what the CEO has to say, won't you?

6.2.1 Setting up the options

In order to use options a class must implement `OptionContext` - in this case we will use a sub-interface of it `VaadinOptionContext`

- Modify `MyNews` to implement `VaadinOptionContext` and implement the stubs of the methods.
- create a constructor and inject `Option` into it
- annotate the constructor with **@Inject**
- return `option` from `optionInstance()`

The result should look like this:

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import com.vaadin.data.Property;
import com.vaadin.ui.Label;
import uk.q3c.krail.core.option.VaadinOptionContext;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.option.Option;

public class MyNews extends Grid3x3ViewBase implements VaadinOptionContext {

    private final Option option;

    @Inject
    protected MyNews(Translate translate, Option option) {
        super(translate);
        this.option = option;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        Label ceoNews = new Label("CEO News");
        Label itemsForSale = new Label("Items for Sale");
        Label vacancies = new Label("Vacancies");
        ceoNews.setSizeFull();
        itemsForSale.setSizeFull();
        vacancies.setSizeFull();
```

(continues on next page)

(continued from previous page)

```
        setMiddleLeft(itemsForSale);
        setCentreCell(ceoNews);
        setMiddleRight(vacancies);
    }

    @Override
    public Option optionInstance() {
        return option;
    }

    @Override
    public void optionValueChanged(Property.ValueChangeEvent event) {

    }
}
```

Options are nothing more than key-value pairs, but we want the keys to be unique across the whole application, and we want them to have a default value so that there is always a value, and, therefore, always predictable behaviour. We will also want them to be presented to users so they can choose a value - which means the option needs a Locale-sensitive name and description. The `OptionKey` provides all of these features.

- define a key for each news channel. They do not have to be public and static, but it can be useful if they are

```
public static final OptionKey<Boolean> ceoVisible = new OptionKey<>(true, MyNews.
↪class, LabelKey.CEO_News_Channel);
public static final OptionKey<Boolean> itemsForSaleVisible = new OptionKey<>(true,␣
↪MyNews.class, LabelKey.Items_For_Sale_Channel);
public static final OptionKey<Boolean> vacanciesVisible = new OptionKey<>(true,␣
↪MyNews.class, LabelKey.Vacancies_Channel);
```

The real key - the one that is used in persistence - is made up of the context, the name key and qualifiers (if used). The context is there to help ensure easily managed uniqueness. Qualifiers are not used in this example, and are only really necessary if you want something like "Push Button 1", "Push Button 2" - you can use the qualifier for the final digit.

Note: An option value is just an object to Krail. Supported data types will be determined by your choice of persistence. However, the core provides a utility class `DataConverter`, to help with the process of translating to String for persistence.

We will make use of these keys in the `optionValueChanged` method, to hide or show the news channels:

- make the `Label` items into fields instead of local variables
- add the code to make the channels visible or hidden depending on the option value

```
@Override
public void optionValueChanged(Property.ValueChangeEvent event) {
    ceoNews.setVisible(option.get(ceoVisible));
    itemsForSale.setVisible(option.get(itemsForSaleVisible));
    vacancies.setVisible(option.get(vacanciesVisible));
}
```

Finally, we need to make sure these options are processed as part of the build, so we call `optionValueChanged` from `doBuild`

```
@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
   super.doBuild(busMessage); ceoNews = new Label("CEO News");
   itemsForSale = new Label("Items for Sale"); vacancies = new Label("Vacancies");
   ceoNews.setSizeFull(); itemsForSale.setSizeFull();
   vacancies.setSizeFull(); setMiddleLeft(itemsForSale);
   setCentreCell(ceoNews); setMiddleRight(vacancies);
   optionValueChanged(null); }
```

Now we have options but we do not have any way of changing them. We will use `OptionPopup` to enable that . . .

Inject `OptionPopup` into the constructor

```
@Inject public MyNews(Option option, OptionPopup optionPopup) {
   this.option = option;
   this.optionPopup = optionPopup;
}
```

- Add a button in `doBuild()` to invoke the popup

```
popupButton=new Button ("options");
popupButton.addClickListener(event->optionPopup.popup(this,LabelKey.News\_Options));
setBottomCentre(popupButton);
```

This is how the whole class should look now:

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.LabelKey;
import com.google.inject.Inject;
import com.vaadin.data.Property;
import com.vaadin.ui.Button;
import com.vaadin.ui.Label;
import uk.q3c.krail.core.option.OptionPopup;
import uk.q3c.krail.core.option.VaadinOptionContext;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.option.Option;
import uk.q3c.krail.option.OptionKey;

public class MyNews extends Grid3x3ViewBase implements VaadinOptionContext {

    public static final OptionKey<Boolean> ceoVisible = new OptionKey<>(true, MyNews.
↪class, LabelKey.CEO_News_Channel);
    public static final OptionKey<Boolean> itemsForSaleVisible = new OptionKey<>(true,
↪ MyNews.class, LabelKey.Items_For_Sale_Channel);
    public static final OptionKey<Boolean> vacanciesVisible = new OptionKey<>(true,␣
↪MyNews.class, LabelKey.Vacancies_Channel);

    private final Option option;
    private final OptionPopup optionPopup;
    private Label ceoNews;
    private Label itemsForSale;
    private Label vacancies;
    private Button popupButton;

    @Inject
```

(continues on next page)

(continued from previous page)

```
    protected MyNews(Translate translate, Option option, OptionPopup optionPopup) {
        super(translate);
        this.option = option;
        this.optionPopup = optionPopup;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        ceoNews = new Label("CEO News");
        itemsForSale = new Label("Items for Sale");
        vacancies = new Label("Vacancies");
        ceoNews.setSizeFull();
        itemsForSale.setSizeFull();
        vacancies.setSizeFull();

        popupButton=new Button("options");
        popupButton.addClickListener(event->optionPopup.popup(this,LabelKey.News_
↪Options));
        setBottomCentre(popupButton);

        setMiddleLeft(itemsForSale);
        setCentreCell(ceoNews);
        setMiddleRight(vacancies);
        optionValueChanged(null);
    }

    @Override
    public Option optionInstance() {
        return option;
    }

    @Override
    public void optionValueChanged(Property.ValueChangeEvent event) {
        ceoNews.setVisible(option.get(ceoVisible));
        itemsForSale.setVisible(option.get(itemsForSaleVisible));
        vacancies.setVisible(option.get(vacanciesVisible));
    }
}
```

- Run the application, and login as user “eq”
- Select the “My News” page
- click on the “options” button

The `OptionPopup` scans the `OptionContext` for `OptionKey` fields and presents them for modification by the user

- Un-check the CEO news (he won’t know, honestly) , and the CEO channel will disappear (you might need to move the popup).
- Logout
- Now log in as user “fb”
- Go to the “My News” page and you will find that the CEO channel is back again - because you are a different user
- logout

- log back in as "eq", and as you would expect, the CEO channel is hidden.

We have demonstrated here that options are associated with users. What we haven't seen is what happens if the system level option changes.

In fact, at the moment there are no system level values defined, so if there is no user level value, then the default coded value is used.

- Still logged in as user "eq", open the options popup and click "Reset to Default" for the CEO channel.
- The "CEO News Channel" checkbox becomes checked, and CEO channel re-appears

This is the expected behaviour - we coded a default value of "true" for the `OptionKey`. Now to demonstrate changing the system level value:

- In `doBuild()`, add a new button, "systemOptionButton", and configure it to change the option value at system level
- We also want to call `optionValueChanged` so we can see the impact of the change
- and of course we need to put the button on the page

```
systemOptionButton = new Button("system option");
systemOptionButton.addClickListener(event -> {
    option.set(ceoVisible, 1, false);
    optionValueChanged(null);
});
setBottomRight(systemOptionButton);
```

- Run the application and login as "eq"
- Navigate to "My News" and you will see that the CEO channel is back - the default `OptionStore` is in-memory, so values are lost when we restart the application
- Try pressing "system option". You will be told that you do not have permission for that action. (There is a bug which presents the stacktrace instead of a user notification)
- Click on the splash message to clear it

We will come to User Access Control in detail later, but for now it is enough to know that `DefaultRealm` - which provides the authorisation rules - allows users to set their own options, but only allows the 'admin' user to set system level options.

- Log out, and log back in as 'admin'. Yes it is the same password.
- Navigate to "My News" and press "system option" again.
- The 'admin' user has permission, so now you will se that the CEO News channel has disappeared.
- press "options" to get the popup, and check "CEO News Channel".
- The item re-appears.
- Press "Reset to Default" for the CEO News Channel and the checkbox is cleared again.

This is demonstrating that the "Override" principle mentioned earlier. If a user has set an option, it is used. If there is no user level value, the system level value is used. Failing that, then the hard code default value is used.

6.3 Using Hierarchies

If you think about it, this hierarchy principle could be used in other scenarios. You could have hierarchies based on geographic location - maybe *city, country, region*. Or another based on job - maybe *function, department, team, role*.

The structure of these may be available from other systems - HR, Identity Management, Facilities systems - or you could define them yourself. You can have as many hierarchies as you wish, and we will come back to this subject later to create a hierarchy of our own.

6.4 Option Data Types

When using the default in memory store, Krail can use any data type for an option. However, most persistence providers will want to confine Option values to a single table, and `DataConverter` provides support for that, by translating `Option` values to `String` and back again.

This supports most primitive data types , `Enum` and `I18NKey`. Collections cannot be used directly, but are supported through `uk.q3c.util.data.collection.DataList`.

`AnnotationOptionList` enables the use of a list of `Annotation` classes.

See `uk.q3c.util.DefaultDataConverter` for the complete list of supported types.

6.5 Summary

We have:

- introduced options, and their purpose
- demonstrated their hierarchical nature
- seen that user access control is applied to options
- shown that `OptionKey` provides a full key definition, enabling the `OptionPopup` to populate without any further coding

6.6 Download from GitHub

To get to this point straight from GitHub:

- git clone https://github.com/davidsowerby/krail-tutorial.git
- cd krail-tutorial
- git checkout –track origin/krail_0.10.0.0

Revert to commit *Options and UserHierarchies Complete*

CHAPTER 7

Configuration from INI files

In the previous section we covered the use of options, and mentioned that Krail sees Options as the top layer of configuration.

This Tutorial has so far covered the top layer and part of the bottom layer. The "bottom layer" includes the configuration we have done using Guice and Guice modules, but includes anything which requires a recompile (for example, annotations)

The middle layer is the one provided by the facility to load ini files (and other formats), and that is what we will explore in this section

7.1 Overview

Krail integrates Apache Commons Configuration to provide support for this form of loading configuration, which extends well beyond just ini files. (See the Apache documentation for more information).

More specifically, Krail captures configuration information in an instance of `ApplicationConfiguration`, which allows a set of configuration values to override a previous set (when they have the same property names). This is similar in principle to the way options work.

7.2 Example

- In the 'pages' package create a new view, 'IniConfigView', extended from `Grid3x3ViewBase`
- Override the `doBuild()` method (you will get some compile errors)

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import com.vaadin.ui.Alignment;
import com.vaadin.ui.Button;
import com.vaadin.ui.Label;
```

(continues on next page)

(continued from previous page)

```
import uk.q3c.krail.config.ApplicationConfiguration;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;

public class IniConfigView extends Grid3x3ViewBase {

[source]
----
private final ApplicationConfiguration applicationConfiguration;
private Label tutorialQualityProperty;
private Label connectionTimeoutProperty;
private Label tutorialCompletedProperty;

@Inject
protected IniConfigView(Translate translate, ApplicationConfiguration␣
↪applicationConfiguration) {
    super(translate);
    this.applicationConfiguration = applicationConfiguration;
}

@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    super.doBuild(busMessage);
    Button showConfigButton = new Button("Show config");
    tutorialQualityProperty = new Label();
    showConfigButton.addClickListener(event -> showConfig());
    setTopCentre(tutorialQualityProperty);
    setMiddleCentre(showConfigButton);
    getGridLayout().setComponentAlignment(tutorialQualityProperty, Alignment.MIDDLE_
↪CENTER);
    connectionTimeoutProperty = new Label();
    tutorialCompletedProperty = new Label();
    setTopRight(tutorialCompletedProperty);
    setTopLeft(connectionTimeoutProperty);

}
private void showConfig() {
    tutorialQualityProperty.setValue("Tutorial quality is: " +␣
↪applicationConfiguration.getString("tutorial.quality"));
    tutorialCompletedProperty.setValue("Tutorial completed: "+␣
↪applicationConfiguration.getString("tutorial.completed"));
    connectionTimeoutProperty.setValue("The timeout is set to: " +␣
↪applicationConfiguration.getString("connection.timeout"));
}
----

}
`
This sets up a button to show the config, and labels to display the values.  Loading␣
↪the config into a singleton instance of`ApplicationConfiguration``` actually␣
↪happens at application startup.

We can inject `ApplicationConfiguration` anywhere in the application to gain access␣
↪to the configuration data loaded from the ini files (or any of the many other␣
↪sources https://commons.apache.org/proper/commons-configuration/[Apache Commons␣
↪Configuration] supports)
```

(continues on next page)

(continued from previous page)

```
Note the `showConfig()` method could equally be placed directly in the lambda␣
↪expression for `showConfigButton`

* Include this new page in `MyOtherPages`
[source]
----
addEntry("ini-config", IniConfigView.class, LabelKey.Ini_Config, PageAccessControl.
↪PUBLIC);
----

= More layers

When an application comprises multiple libraries, there may be a need for multiple␣
↪sets of configuration. You can add as many configuration files as you require.

== Adding ini files

* create a file 'krail.ini' in _src/main/webapp/WEB-INF_
* you may be familiar with the extended properties file format .... populate it with:
`
[tutorial]
quality=good
completed=false
`
*

create another file 'moreConfig.ini' in WEB-INF, with this content:
```

quality=brilliant

timeout=1000

```
This will be used to show a property overriding another, while also adding new␣
↪properties.

== Configure Guice

We now need to set up the Guice configuration so it knows about the additional file.␣
↪You can sub-class `ApplicationConfigurationModule` , and then tell the␣
↪`BindingManager` about it, or more easily, simply add the configs as part of the␣
↪the `BindingManager` entry like this:

[source]
----
    @Override
    protected Module applicationConfigurationModule() {
        return new KrailApplicationConfigurationModule().addConfig("moreConfig.ini",
↪98,false).addConfig("krail.ini",100,true);
    }
----

Be aware that the order that the files are processed is important if they contain the␣
↪same (fully qualified) property names. If you look at the javadoc for `addConfig()`␣
↪you will see that the second parameter determines the order (priority) of loading,␣
↪with a lower value being the highest priority (0 is therefore the highest priority)
```

(continues on next page)

(continued from previous page)

```
* Run the application and select the "Ini Config" page
*

Press "Show config" and you will see the values provided by _krail.ini_ and _
↪moreConfig.ini_ combined:

** _tutorial.completed_ from _krail.ini_ is unchanged as there is no value for it in _
↪moreConfig.ini_
** _connection.timeout_ is a new property from _moreConfig.ini_
** _tutorial.quality_ from _krail.ini_ has been overridden by the value in _
↪moreConfig.ini_

= Fail early

If an ini file is essential for the operation of your application, `addConfig()`␣
↪allows you to specify that. Both the examples have the 'optional' parameter set to
↪'false', but of course both files are present.

* add another config to the `BindingManager entry`, but do not create the␣
↪corresponding file
`
@Override
protected Module applicationConfigurationModule() {
    return new KrailApplicationConfigurationModule()
            .addConfig(&quot;moreConfig.ini&quot;,98,false)
            .addConfig(&quot;essential.ini&quot;,99,false)
            .addConfig(&quot;krail.ini&quot;,100,true);
}
`
* run the application and it will fail early with a `FileNotFoundException` (Note:␣
↪there is currently a https://github.com/davidsowerby/krail/issues/531[bug] which␣
↪causes a timeout rather than an exception)
* change the 'optional' parameter to true and the application will run
[source]
----
    @Override
    protected Module applicationConfigurationModule() {
        return new KrailApplicationConfigurationModule()
                .addConfig("moreConfig.ini",98,false)
                .addConfig("essential.ini",99,false)
                .addConfig("krail.ini",100,false);
    }
----

The final versions of the files should be:

[source]
----
package com.example.tutorial.app;

import com.example.tutorial.i18n.LabelKey;
import com.example.tutorial.pages.AnnotatedPagesModule;
import com.example.tutorial.pages.MyOtherPages;
import com.example.tutorial.pages.MyPages;
import com.example.tutorial.pages.MyPublicPages;
import com.google.inject.Module;
import uk.q3c.krail.core.config.KrailApplicationConfigurationModule;
```

(continues on next page)

(continued from previous page)

```
import uk.q3c.krail.core.guice.DefaultServletContextListener;
import uk.q3c.krail.core.navigate.sitemap.SystemAccountManagementPages;
import uk.q3c.krail.core.sysadmin.SystemAdminPages;
import uk.q3c.krail.core.ui.DefaultUIModule;

import java.util.List;

public class BindingManager extends DefaultBindingManager {

    @Override
    protected Module servletModule() {
        return new TutorialServletModule();
    }

    @Override
    protected void addAppModules(List<Module> modules) {

    }

    @Override
    protected void addSitemapModules(List<Module> baseModules) {
        baseModules.add(new SystemAccountManagementPages());
        baseModules.add(new MyPages().rootURI("private/finance-department"));
        baseModules.add(new AnnotatedPagesModule());
        baseModules.add(new SystemAdminPages());
        baseModules.add(new MyPublicPages());
        baseModules.add(new MyOtherPages());
    }

    @Override
    protected Module uiModule() {
        return new DefaultUIModule().uiClass(TutorialUI.class).
↪applicationTitleKey(LabelKey.Krail_Tutorial);
    }

    @Override
    protected Module applicationConfigurationModule() {
        return new KrailApplicationConfigurationModule()
                .addConfig("moreConfig.ini",98,false)
                .addConfig("essential.ini",99,true)
                .addConfig("krail.ini",100,true);
    }
}

----

[source]
----
package com.example.tutorial.pages;

import com.google.inject.Inject;
import com.vaadin.ui.Alignment;
import com.vaadin.ui.Button;
import com.vaadin.ui.Label;
import uk.q3c.krail.config.ApplicationConfiguration;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
```

(continues on next page)

(continued from previous page)

```
import uk.q3c.krail.i18n.Translate;

public class IniConfigView extends Grid3x3ViewBase {

    private final ApplicationConfiguration applicationConfiguration;
    private Label tutorialQualityProperty;
    private Label connectionTimeoutProperty;
    private Label tutorialCompletedProperty;

    @Inject
    protected IniConfigView(Translate translate, ApplicationConfiguration␣
↪applicationConfiguration) {
        super(translate);
        this.applicationConfiguration = applicationConfiguration;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        Button showConfigButton = new Button("Show config");
        tutorialQualityProperty = new Label();
        showConfigButton.addClickListener(event -> showConfig());
        setTopCentre(tutorialQualityProperty);
        setMiddleCentre(showConfigButton);
        getGridLayout().setComponentAlignment(tutorialQualityProperty, Alignment.
↪MIDDLE_CENTER);
        connectionTimeoutProperty = new Label();
        tutorialCompletedProperty = new Label();
        setTopRight(tutorialCompletedProperty);
        setTopLeft(connectionTimeoutProperty);

    }
    private void showConfig() {
        tutorialQualityProperty.setValue("Tutorial quality is: " +␣
↪applicationConfiguration.getString("tutorial.quality"));
        tutorialCompletedProperty.setValue("Tutorial completed: "+␣
↪applicationConfiguration.getString("tutorial.completed"));
        connectionTimeoutProperty.setValue("The timeout is set to: " +␣
↪applicationConfiguration.getString("connection.timeout"));
    }
}
----

= Summary

* We have loaded an ini file
* we have demonstrated the principle of overriding the the values in one ini file␣
↪with those from another
* We have demonstrated ensuring an early fail if a file is missing
* We have demonstrate making the presence of an ini file optional

Apache Commons Configuration supports much more than just ini files, and can support␣
↪https://commons.apache.org/proper/commons-configuration/userguide_v1.10/overview.
↪html#Configuration_Sources[variety of sources] - Krail will just accept anything␣
↪that Apache Commons Configuration provides

= Download from GitHub
```

(continues on next page)

(continued from previous page)

```
To get to this point straight from GitHub:

[source,bash]
----
git clone https://github.com/davidsowerby/krail-tutorial.git
cd krail-tutorial
git checkout --track origin/krail_0.10.0.0

----

Revert to commit _Configuration from ini file complete_
```

CHAPTER 8

User Access Control

You have seen some aspects of Krail's User Access Control already, and are probably aware that it provides this by integrating Apache Shiro. This Tutorial will not attempt to cover the whole of Shiro's capability - Shiro's own documentation does a good job of that already.

What we will do, however, is demonstrate some of the features of Shiro, within a Krail context:

- **Implementing a Realm**. Implement a trivial Realm to provide authentication and authorisation
- **Page Access Control**. This is Krail specific use of Shiro features to determine whether a user has permission to access a page
- **Coded access**. Checking from code whether a user has permissions to do something
- **Access Control annotations**. This will demonstrate the use of Shiro's annotations, as an alternative to using coded access

Krail does not yet provide any user management capability (the management of users, groups & roles etc) as this is often provided via LDAP, Active Directory or Identity Management systems. There is an open ticket for it, so it may be developed one day.

8.1 Example

We will take this opportunity to tidy up our site, and limit who can use different parts of the site. This is what we want to achieve:

- the 'finance' pages should be on their own branch
- public pages will remain available to any user
- private pages will be limited to just 2 users, "eq" and "fb"
- both users will have access to the 'private' branch
- both users will be able to change their own options
- 'fb' will be able to access the finance pages, but 'eq' will not.

- there will be an 'admin' user who can access all pages and change all options

At this point we must stress that this is going to be a trivial example of User Access Control, and to do it properly you need to consult the Shiro documentation. This Tutorial should give you some useful pointers, however.

8.2 Move the Pages

To move the 'finance' pages:

- change the line in the `BindingManager` to put the `MyPages` root URI at *finance* instead of *private/finance-department*

```
baseModules.add(new MyPages().rootURI("finance"));
```

- In the `PurchasingView` change the uri parameter to be *finance/purchasing*

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.core.navigate.sitemap.View;
import uk.q3c.krail.core.shiro.PageAccessControl;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.i18n.Translate;

@View(uri = "finance/purchasing", pageAccessControl = PageAccessControl.PERMISSION,␣
↪labelKeyName = "Purchasing")
public class PurchasingView extends Grid3x3ViewBase {

    @Inject
    protected PurchasingView(Translate translate) {
        super(translate);
    }
}
```

- In `NewsView.doBuild()` change the button event to point to the new page location ' navigateToPrivatePage.addClickListener(c -> navigator.navigateTo("finance/accounts")); '

8.3 User accounts

- create a new package, 'com.example.tutorial.uac'
- in that package create a new class "TrivialUserAccount" - it is obvious what it does

```
package com.example.tutorial.uac;

import java.util.Arrays;
import java.util.List;

public class TrivialUserAccount {

[source]
----
private String password;
private List<String> permissions;
```

(continues on next page)

(continued from previous page)

```
private String userId;

public TrivialUserAccount(String userId, String password, String... permissions) {
    this.userId = userId;
    this.password = password;
    this.permissions = Arrays.asList(permissions);
}

public String getUserId() {
    return userId;
}

public String getPassword() {
    return password;
}

public List<String> getPermissions() {
    return permissions;
}
----

}
```

 Note You may notice that there is no "role" in this user account. You can certainly use Shiro's roles in Krail, but we prefer to use permissions for the reasons given by the Shiro team.

8.4 Credentials Store

- create a class ""TrivialCredentialsStore" as somewhere to keep the user accounts:

```
package com.example.tutorial.uac;

import com.google.inject.Inject;

import java.util.HashMap;
import java.util.Map;

public class TrivialCredentialsStore  {
    private Map<String, TrivialUserAccount> store = new HashMap<>();

    @Inject
    protected TrivialCredentialsStore() {
    }

    public TrivialCredentialsStore addAccount(String userId, String password, String..
↪. permissions) {
        store.put(userId, new TrivialUserAccount(userId, password, permissions));
        return this;
    }

    public TrivialUserAccount getAccount(String principal) {
        return store.get(principal);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

- define the users' credentials to meet our requirements - we'll just put them in the constructor

```
@Inject
protected TrivialCredentialsStore() {
    addAccount("eq", "eq", "page:view:private:*",
↪"option:edit:SimpleUserHierarchy:eq:0:*:*");
    addAccount("fb", "fb", "page:view:private:*","page:view:finance:*",
↪"option:edit:SimpleUserHierarchy:fb:0:*:*");
    addAccount("admin", "password", "page:view:*","option:edit:*");
}
```

8.4.1 Permission Strings

What we have done here is give users specific credentials. The userId and password are obvious. The permission strings use Shiro's `WildcardPermission`.

This is a very flexible way of defining permissions. Krail uses the `WildcardPermission` to define page and option approval.

8.4.2 Page Permission

So for example, a page with a url of:

```
private/apage/asubpage/id=1
```

is translated by Krail's `PagePermission` into a Shiro compatible syntax of:

```
page:view:private:apage:asubpage
```

This represents:

- resource type ('page')
- action ('view')
- resource instance (the Url with the '/' transposed to a ':' to match the Shiro syntax)
- the url parameter is ignored, because it is not part of the page definition

This is then compared, by Shiro, with the permission a user has been given. Both 'eq' and 'fb' have been given a permission: ' page:view:private:* ' which translates to "for a resource type page, this user can view any with a url starting with *private*"

The 'admin' user has been given permission to view any page, simply by wildcarding all pages

```
page:view:*
```

8.4.3 Option permission

An Option follows a similar pattern, provided by `OptionPermission`

- resource type ('option')

- action ('edit')
- resource instance (an option) structured [hierarchy]:[user id]:[hierarchy level index]:[context]:[option name]:[qualifier]:[qualifier]

Thus the option permissions given to 'eq' and 'fb' only allow them to edit their own options in the `SimpleUserHierarchy`. This is set by giving permission only at the user level, hierarchy level index = 0

Again the 'admin' user is all-powerful, with permission to edit any option:

```
option:edit:*
```

8.5 Authentication

Shiro has the concept of a `Realm`, where the rules for Authentication and Authorisation are defined - by you, as they will be application specific. Shiro offers a number of ways to implement Realm, and here we will just provide a trivial example, combining authentication and authorisation into one `Realm`

We will sub-class `AuthorizingRealmBase`, as that provides a mechanism for enabling the cache via Guice.

- in the package, 'com.example.tutorial.uac' create a class "TutorialRealm", extending `AuthorizingRealmBase`

```
package com.example.tutorial.uac;

import uk.q3c.krail.core.shiro.AuthorizingRealmBase;

public class TutorialRealm extends AuthorizingRealmBase {

}
```

- We want to use our `TrivialCredentialsStore`, so we will inject that into the constructor
- Caching obviously is not needed for this trivial case, but we will pass `Optional<CacheManager>` to `AuthorizingRealmBase`. This will allow us to demonstrate enabling the cache from Guice.

```
public class TutorialRealm extends AuthorizingRealmBase {

    private TrivialCredentialsStore credentialsStore;

    @Inject
    protected TutorialRealm(Optional<CacheManager> cacheManagerOpt,␣
↪TrivialCredentialsStore credentialsStore) {
        super(cacheManagerOpt);
        this.credentialsStore = credentialsStore;
    }
}
```

- provide the authentication logic by overriding `doGetAuthenticationInfo()`

```
@Override
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)␣
↪throws AuthenticationException {
    TrivialUserAccount userAccount = credentialsStore.getAccount((String) token.
↪getPrincipal());
    if (userAccount == null) {
        return null;
```

(continues on next page)

(continued from previous page)

```
    }
    String tokenCredentials = new String((char[])token.getCredentials());
    if(userAccount.getPassword().equals(tokenCredentials)) {
        return new SimpleAuthenticationInfo(userAccount.getUserId(),token.
↪getCredentials(),"TutorialRealm");
    }else{
        return null;
    }
}
```

This logic returns null if the user account is not found, or the password supplied by the token does not match the credentials. If authentication is successful, a populated instance of `SimpleAuthenticationInfo` is returned

8.6 Authorisation

- override `doGetAuthorizationInfo()` to provide the authorisation logic

```
@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
    TrivialUserAccount userAccount = credentialsStore.getAccount((String) principals.
↪getPrimaryPrincipal());
    if (userAccount != null) {
        SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();
        info.setStringPermissions(new HashSet<>(userAccount.getPermissions()));
        return info;
    }
    return null;
}
```

This logic returns a populated `SimpleAuthorizationInfo` instance if the user account is found, or null if not

8.7 Using the Realm

- override the `shiroModule()` method in the `BindingManager` to use the new `Realm`
- enable the cache as shown

```
@Override
protected Module shiroModule() {
    return new DefaultShiroModule().addRealm(TutorialRealm.class).enableCache();
}
```

*

run the application and check to see if we have met our requirements:

- log in as 'eq', with password 'eq'
- *private* pages should be visible, but not the *finance* pages or *system admin* pages
- you should still be able to modify options on the "My News" page
- pressing the "system option" button on "My News" will result in a "You do not have permission" message
- log out

- log in as 'fb' - try a wrong password if you like, the correct password should be 'fb'
- *private* and *finance* pages should be visible, but not *system admin* pages
- you should still be able to modify options on the "My News" page
- pressing the "system option" button on "My News" will result in a "You do not have permission" message
- log out
- log in as 'admin', password= 'password'
- *private*, *finance* and *system admin* pages pages should all be visible
- you should still be able to modify options on the "My News" page
- pressing the "system option" button on "My News" remove the CEO news

So far this has all been done using page and option permissions. The visibility of pages is actually managed through `PageAccessControl` which limits what is made available to the navigation components. You can take also direct control using code or Shiro annotations.

8.8 Control Access Through Code

At the moment the "system option" button on "My News" can result in a "You do not have permission" message. It does not make much sense to make the button available to a user who is not allowed to use it, so let's hide the button unless the user has permission.

- to get access to the current Shiro `Subject`, we inject a `SubjectProvider`
- modify `MyNews` to do so:

```
@Inject
public MyNews(Option option, OptionPopup optionPopup, SubjectProvider subjectProvider,
↪ Translate translate) {
    super(translate);
    this.option = option;
    this.optionPopup = optionPopup;
    this.subjectProvider = subjectProvider;
}
```

- in `MyNews.doBuild()` make the visibility conditional on the user having permission

```
if (subjectProvider.get().isPermitted("option:edit:SimpleUserHierarchy:*:1:*:*")) {
    systemOptionButton.setVisible(true);
}else{
    systemOptionButton.setVisible(false);
}
```

Here we have asked Shiro to confirm permission at the most specific level, as recommended by Shiro. This permission string is checking that the user has permission to edit any option at level 1 (the 'system' level) in the `SimpleUserHierarchy`

- run the application and log in as 'eq' or 'fb' and you will not be able to see the "system option" button. Log in as 'admin', however, and the "system option" button is visible.

8.9 Control Access Through Annotations

Shiro provides a set of annotations to cover most circumstances. We will use **@RequiresPermissions** as an example

- on the `MyNews` page add another button in `doBuild()`

```
payRiseButton = new Button("request a pay rise");
payRiseButton.addClickListener(event-> requestAPayRise());
setBottomLeft(payRiseButton);
```

- inject the `UserNotifier`

```
@Inject
public MyNews(Option option, OptionPopup optionPopup, SubjectProvider subjectProvider,
↪ Translate translate, UserNotifier userNotifier) {
    super(translate);
    this.option = option;
    this.optionPopup = optionPopup;
    this.subjectProvider = subjectProvider;
    this.userNotifier = userNotifier;
}
```

- create the `requestAPayRise` method
- use `userNotifier` to give feedback
- create the enum constant **DescriptionKey.You_just_asked_for_a_pay_increase**

```
protected void requestAPayRise() {
    userNotifier.notifyInformation(DescriptionKey.You_just_asked_for_a_pay_increase);
}
```

- We want to restrict who can use the method, so we will annotate it with a new permission

```
@RequiresPermissions("pay:request-increase")
protected void requestAPayRise() {
    userNotifier.notifyInformation(DescriptionKey.You_just_asked_for_a_pay_increase);
}
```

Nobody currently has permission to do this, so let's allow user 'eq' to do this

- modify the entry for 'eq' in `TrivialCredentialsStore` to add this permission

```
addAccount("eq", "eq", "page:view:private:*","option:edit:SimpleUserHierarchy:eq:0:*:*
↪","pay:request-increase");
```

- run the application
  - log in as 'eq'
  - navigate to "My News" and press "request a pay rise".
  - A notification pops up to confirm the request. (Unfortunately it doesn't say what will happen to the request)
  - log in as 'fb' or 'admin'
  - navigate to "My News" and press "request a pay rise".
  - you receive a "not permitted" message

8.10 Summary

We have: - Shown how to control access to pages
 - Shown how access control is applied to Options
 - Shown how to control access using code
 - Shown how to control access using annotations
 - Built a very simple credential store with user accounts
 - Demonstrated some uses of Shiro's Wildcard permissions

8.11 Download from GitHub

To get to this point straight from GitHub:

```
git clone https://github.com/davidsowerby/krail-tutorial.git
cd krail-tutorial
git checkout --track origin/krail_0.10.0.0
```

Revert to commit *User Access Control Complete*

CHAPTER 9

I18N

We had a very brief introduction to Krail's implementation of I18N in an earlier section of the Tutorial, and this gave us the `LabelKey` and `DescriptionKey` **enum** classes.

In this section we will cover most of the rest of Krail's I18N functionality.

9.1 Elements of I18N

A complete I18N "transaction" requires the following components:

- a key to identify a pattern
- Locale-specific patterns for as many languages you wish to support
- arguments to populate variables in the pattern (if there are any)
- a method for selecting the the correct language and applying the arguments to the pattern
- a way of knowing which Locale to use

9.2 Direct translation

Open up `MyNews` and you will recall that we have used `String` literals in a number of places. This is going to make life difficult if ever we want to translate this application - and even if we just want to use the same phrase in a number of different places.

Let's replace the following literal (in `doBuild()` ') with something more robust:

```
popupButton = new Button("options");
```

with

```
popupButton = new Button(getTranslate().from(LabelKey.Options));
```

- create the 'Options' enum constant in `LabelKey`

This simple step gives you:

1. refactoring support for the key
2. a default translation, which is the enum's `name()` method with underscores transposed to spaces.
3. a way to provide an alternative phrase, without changing the enum key (we will see that shortly)
4. an application which requires no code changes if additional language support is needed one day

9.2.1 Message with Parameters

Now let us add a banner to the page, which will include some variable information.

- in `doBuild()` add: ' Label bannerLabel = new Label(); getGridLayout().addComponent(bannerLabel,0,0,1,0); '

So far, all the I18 patterns have been simple - they have had no parameters. Now we want a more complex message with some dynamic elements to it.

At this stage, you may want to consider a convention for naming your keys. In general we feel it is best to group them with a feature, and then perhaps consider splitting them further:

- Labels : short, usually one or two words, no parameters, generally used as captions
- Descriptions : longer, typically several words, no parameters, generally used in tooltips
- Messages : contains parameter(s).

This is just a convention - which we will use in this Tutorial - but it is entirely your decision how you organise your keys.

To stick to this convention, we will now rename the key classes we have already created:

- Rename `LabelKey` to `TutorialLabelKey`
- Rename `DescriptionKey` to `TutorialDescriptionKey`

Creating a Key class

Assuming you have followed this Tutorial from the start, you have already seen how to create a key class. We are going to add another now:

- in the 'com.example.tutorial.i18n' package, create an Enum class called 'TutorialMessageKey'. It should implement the `I18NKey` interface
- create a `TutorialMessageKey` constant **Banner**

```
package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.I18NKey;

public enum TutorialMessageKey implements I18NKey {
    Banner
}
```

This is going to be a long message, and because it has parameters, the default translation cannot be taken from the key name. We will use a class based method for defining the pattern:

- in the 'com.example.tutorial.i18n' package, create a class 'TutorialMessages' (the naming convention is important here - in order to find values for keys, the default is to assume, for example, that 'LabelKey' will map to 'Labels' - although this can be changed by overriding `I18NKey.bundleName()`
- override the `loadMap()` method

```
package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.EnumResourceBundle;

public class TutorialMessages extends EnumResourceBundle<TutorialMessageKey>{
 @Override
 protected void loadMap() {

[source]
----
}
----

}
```

Here you will see that we are extending `EnumResourceBundle` but for type safety, genericised with `TutorialMessageKey`. The `loadMap()` method enables entries to be put in a map.

- now associate the **Banner** key with an I18N pattern - using a static import makes it more readable:

```
package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.EnumResourceBundle;

import static com.example.tutorial.i18n.TutorialMessageKey.*;

public class TutorialMessages extends EnumResourceBundle<TutorialMessageKey> {

    @Override
    protected void loadMap() {
        put(Banner,"The temperature today is {1}.  The CEO has noticed that her news␣
↪channel {0}.");
    }
}
```

Each of the parameters - *{n}* - will take a value we supply as an argument. The arguments:

1. must be supplied in the order of the numbers in the *{n}*, not the order in which they appear in the pattern (because different languages may require parameters in a different order).
2. must match the number of parameters. If not, the whole translation is abandoned and the pattern string is returned unchanged. (**Note:** This is the default behaviour of `Translate`), but as of Krail 0.10.0.0 `Translate` offers different levels of "strictness" regarding the matching of parameters to arguments. See the javadoc for detail.

```
Now let's display the banner:

- set up a random temperature
- choose a key depending on whether the CEO News channel is selected
- add two keys to ```TutorialLabelKey```, **is_selected** and **is_not_selected**
- create a ```Label``` using the translated message with the two arguments (remember␣
↪that 'temperature' is the second parameter, *{1}* in the pattern, even though it␣
↪appears first).  In the ```doBuild``` method of ```MyNews``` add:
```

```
int temperature = (new Random().nextInt(40))-10;
TutorialLabelKey selection = (option.get(ceoVisible)) ? TutorialLabelKey.is_selected␣
↪: TutorialLabelKey.is_not_selected;

Label bannerLabel = new Label(getTranslate().from(TutorialMessageKey.Banner, ␣
↪selection, temperature));
getGridLayout().addComponent(bannerLabel,0,0,2,0);
```

```
Parameters passed as ```I18NKey``` constants are also translated.  These are␣
↪currently the only parameter types that are localised, see [open ticket](https://
↪github.com/davidsowerby/krail/issues/428).

- Run the application, log in and and navigate to "MyNews" (login = 'eq', 'eq'),
    - the banner has been expanded to include the variable values
- click on "options" and change the value for CEO New Channel - but the label does␣
↪not change, because the banner has no way of knowing the option value has changed.
- To fix this
    - make ```bannerLabel``` a field
    - move the code to set the bannerLabel value to ```optionValueChanged```
    - move 'optionValueChanged(null);' to the end of ```doBuild()```


The full code for ```doBuild()``` method is now:
```

```
@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    super.doBuild(busMessage);
    ceoNews = new Label("CEO News");
    itemsForSale = new Label("Items for Sale");
    vacancies = new Label("Vacancies");
    ceoNews.setSizeFull();
    itemsForSale.setSizeFull();
    vacancies.setSizeFull();

    popupButton = new Button(getTranslate().from(TutorialLabelKey.Options));
    popupButton.addClickListener(event -> optionPopup.popup(this, TutorialLabelKey.
↪News_Options));
    setBottomCentre(popupButton);

    systemOptionButton = new Button("system option");
    systemOptionButton.addClickListener(event -> {
        option.set(ceoVisible, 1, false);
        optionValueChanged(null);
    });
    setBottomRight(systemOptionButton);

    setMiddleLeft(itemsForSale);
    setCentreCell(ceoNews);
    setMiddleRight(vacancies);

    if (subjectProvider.get().isPermitted("option:edit:SimpleUserHierarchy:*:1:*:*"))
↪{
        systemOptionButton.setVisible(true);
    } else {
        systemOptionButton.setVisible(false);
    }
```

(continues on next page)

(continued from previous page)

```
    payRiseButton = new Button("request a pay rise");
    payRiseButton.addClickListener(event -> requestAPayRise());
    setBottomLeft(payRiseButton);

    bannerLabel = new Label();
    getGridLayout().addComponent(bannerLabel,0,0,2,0);

    optionValueChanged(null);

}
```

```
```optionValueChanged()``` is now:
```

```
@Override
public void optionValueChanged(Property.ValueChangeEvent event) {
    ceoNews.setVisible(option.get(ceoVisible));
    itemsForSale.setVisible(option.get(itemsForSaleVisible));
    vacancies.setVisible(option.get(vacanciesVisible));
    int temperature = (new Random().nextInt(40)) - 10;
    TutorialLabelKey selection = (option.get(ceoVisible)) ? TutorialLabelKey.is_
↪selected : TutorialLabelKey.is_not_selected;
    bannerLabel.setValue(getTranslate().from(TutorialMessageKey.Banner, selection,␣
↪temperature));
}
```

```
- Rerun the application, login and select 'My News' page, and try changing the option␣
↪to display the CEO new channel.  The banner will update to demonstrate that she␣
↪really is watching you ...

#Translation from Annotations

When using Vaadin components, it is often more convenient to use an ```Annotation```␣
↪instead of calling ```Translate``` directly - this keeps the ```I18NKey```s with␣
↪the fields using them.

To achieve this, we need an annotation that is specific to our ```I18NKey```␣
↪implementations (we cannot use annotations from Krail core, because of the␣
↪limitations Java places on ```Annotation``` parameters)

- in the package 'com.example.tutorial.i18n', create a new Annotation class called
↪"TutorialCaption".  Note the ```@I18NAnnotation``` - this tells Krail's␣
↪```I18NAnnotationProcessor``` that this annotation is used for I18N.
```

package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.I18NAnnotation;

import java.lang.annotation.ElementType; import java.lang.annotation.Retention; import java.lang.annotation.RetentionPolicy; import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) @Target({ElementType.FIELD, ElementType.TYPE}) @I18NAnnotation public @interface TutorialCaption {

```
TutorialLabelKey caption();

TutorialDescriptionKey description();
```
```

}

```
The annotation itself can be called anything, but it must be annotated␣
↪`@I18NAnnotation`, and its methods be one or more of:

. `I18NKey implementation` caption() - _used for component captions_
. `I18NKey implementation` description() - _used for component tooltips_
. `I18NKey implementation` value() - _used where a component implements the Property␣
↪interface, typically Label_
. `String locale()` - _a locale String to force use of a specific locale for this␣
↪annotation_ (see `Locale.toLanguageTag()` for the String format)

You may combine these methods in any way you wish - Krail's I18N annotation scanner␣
↪(`I18NProcessor`) just looks for annotations which are annotated with␣
↪`@I18NAnnotation` and for any methods in them which match those listed above.

* remove the translate method from the construction of `popupButton` in `doBuild()`
`
popupButton = new Button();
`
* replace it by annotating the `popupButton` field
`
@TutorialCaption(caption = TutorialLabelKey.Options,description=␣
↪TutorialDescriptionKey.Select_your_options)
private Button popupButton;
`
* create the constant for `TutorialDescriptionKey`

Before we run the app, let's also use update the name key for the View

* update the constructor:
[source]
----
    @Inject
    public MyNews(Option option, OptionPopup optionPopup, SubjectProvider␣
↪subjectProvider, Translate translate, UserNotifier userNotifier) {
        super(translate);
        nameKey = TutorialLabelKey.My_News;
        this.option = option;
        this.optionPopup = optionPopup;
        this.subjectProvider = subjectProvider;
        this.userNotifier = userNotifier;
    }
----

The need to do this manually should be fixed by this https://github.com/davidsowerby/
↪krail/issues/625[open issue].

*

Run the application, log in and and navigate to "MyNews" (login = 'eq', 'eq')

** The "Options" button will be the same as before, but of course the caption is␣
↪generated by the annotation
** The tooltip for the "Options" button will now say "Select your options"
** The tab should now say "Krail Tutorial My News"

== Limitations
```

(continues on next page)

(continued from previous page)

```
Naturally, you cannot use variable values with an annotation - by its very nature,␣
↪`Annotation` will only take static values. For I18N patterns which requires dynamic␣
↪values, therefore, you will need to use a direct call to `Translate`.

= Multi-Language

Even though Krail's approach to handling I18N is actually very useful even in a␣
↪single language application, the whole point of I18N is, of course, to support␣
↪multiple languages / Locales.

By default, `I18NModule` defaults everything to *Locale.UK*. This section assumes␣
↪that you are familiar with the standard Java approach to I18N. For those not␣
↪familiar with it, there are many online resources if you need them.
<a name="config-methods"></a>

== Methods of configuration

Krail uses the `I18NModule` to configure how I18N operates. There are two fundamental␣
↪ways to define that configuration (as with most modules):

. Use fluent methods provided by the module, to use at the point of construction in␣
↪the `BindingManager`.
. Sub-class `I18NModule`and use the sub-class in the `BindingManager`

It really does not matter which method you use. We will use method 2 for this example,
↪ but then show how method 1 would achieve the same result, but not actually apply␣
↪it.

* in the package 'com.example.tutorial.i18n', create a new class 'TutorialI18NModule'␣
↪extending from `I18NModule`
* override the `define()` method
```java
package com.example.tutorial.i18n;

import uk.q3c.krail.core.i18n.KrailI18NModule;

public class TutorialI18NModule extends KrailI18NModule {
}
`
- override the`define()``` method to define everything we need to.
- set the default locale explicitly, and add another Locale that we want to support.␣
↪(The default locale is automatically a supported locale)

[source]
----
@Override
    protected void define() {
        defaultLocale(Locale.UK);
        supportedLocales(Locale.GERMANY);
    }
}
----

* use the new class in `BindingManager`
`
@Override
```

(continues on next page)
```

(continued from previous page)

```
protected Module i18NModule() {
return new TutorialI18NModule();
}
`
* in the package 'com.example.tutorial.i18n', create an new class 'TutorialMessages_de
↪' extended from `TutorialMessages`
[source]
----
package com.example.tutorial.i18n;

import static com.example.tutorial.i18n.TutorialMessageKey.Banner;

public class TutorialMessages_de extends TutorialMessages {
    @Override
    protected void loadMap() {
        put(Banner, "Die Temperatur ist heute {1}. Der CEO hat bemerkt, dass ihre␣
↪Nachrichten-Kanal {0}");
    }
}

----

To translate the keys used for parameter _{0}_ we need to do the same for␣
↪`TutorialLabelKeys` - but do not have a `TutorialLabels` class - so far, all␣
↪translation defaulted to the key name.

* create a new class 'TutorialLabels', extended from `EnumResourceBundle`
* implement `loadMap()`
[source]
----
package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.EnumResourceBundle;

public class TutorialLabels extends EnumResourceBundle<TutorialLabelKey>{
    @Override
    protected void loadMap() {

    }
}
----

* create a new class 'TutorialLabels_de' extended from `TutorialLabels`
* put the translations into the map
```

package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.EnumResourceBundle;

import static com.example.tutorial.i18n.TutorialLabelKey.*;

public class TutorialLabels_de extends EnumResourceBundle<TutorialLabelKey>{ @Override protected void loadMap() { put(is_selected, “aktiviert ist”); put(is_not_selected, “nicht aktiviert ist”); put(Options, “die Optionen”); } }

```
- run the application, and:
 - in the Locale selector, top right of the page, select "Deutsch" (the selector␣
↪takes its selection list from the supported locales you have defined)
```

(continues on next page)

(continued from previous page)

```
 - a popup will inform you, in German, of the change
 - a number, but not all items have changed language (Krail has some translations␣
↪built in, and these are the ones which have changed. Hopefully, the number of␣
↪translations will increase over time - if you can contribute, please do)
 - log in and navigate to 'MyNews'
 - most of the page will still be in English (we have not provided translations for␣
↪it all) but the banner and Options button should now be in German.
 - change the language back to English - and the banner stays in German, while the␣
↪Options button switches back to English.

Why is this happening? Well, currently there is nothing to tell this view that it␣
↪should re-write the banner when there is a change in language. The *@Caption*␣
↪annotation handles that automatically, but for a manual translation we need to␣
↪respond to a language change message.

* move the logic for populating the banner to its own method
`
private void populateBanner() {
int temperature = (new Random().nextInt(40)) - 10;
TutorialLabelKey selection = (option.get(ceoVisible)) ? TutorialLabelKey.is_selected␣
↪: TutorialLabelKey.is_not_selected;
bannerLabel.setValue(getTranslate().from(TutorialMessageKey.Banner, selection,␣
↪temperature));
}
`
* `optionValueChanged()` should now look like this
[source]
----
@Override
public void optionValueChanged(Property.ValueChangeEvent event) {
    ceoNews.setVisible(option.get(ceoVisible));
    itemsForSale.setVisible(option.get(itemsForSaleVisible));
    vacancies.setVisible(option.get(vacanciesVisible));
    populateBanner();
}
----

= CurrentLocale and responding to change

You have been using `CurrentLocale` without being aware of it - `Translate` refers to␣
↪it when a call is made to `Translate.from()`. A little explanation is now needed.

[source,CurrentLocale``` holds the currently selected locale for a user.  It is first␣
↪populated from a combination of things like Web Browser settings, and whatever you␣
↪have defined in the ```KrailI18NModule``` - the logic is in described in the␣
↪```DefaultCurrentLocale``` javadoc.]
----

When a change is made to the current locale (in our case, using the␣
↪```LocaleSelector```), ```CurrentLocale``` publishes a ```LocaleChangeBusMessage```␣
↪via the session [Event Bus](tutorial-event-bus.md).  We need to intercept that␣
↪message, and respond to it by updating the banner.

- make this View an event bus listener and subscribe to the session Event Bus

----
```

(continues on next page)

(continued from previous page)

```
@Listener @SubscribeTo(SessionBus.class)
public class MyNews extends Grid3x3ViewBase implements OptionContext {
`
- register a handler for the message - the annotation and the message type are the␣
↪important parts - the method can be called anything
- call`populateBanner`to update its text
`
@Handler
protected void localeChanged(LocaleChangeBusMessage busMessage) {
 populateBanner();
}
```

- Run the application, log in and navigate to 'MyNews'
- Changing locale now immediately updates the banner

9.3 Pattern sources

So far we have used the class-based method for defining I18N patterns. Krail originally supported the traditional properties files, but that has now been withdrawn as we saw no benefit to using it.

You can, however, use any source - a database, REST service or any other service which can provide patterns via a pluggable DAO. Through Guice configuration, each source is identified by an annotation. Krail provides an in-memory map as a source, annotated with **@InMemory**. Being in memory, it is not very useful except for testing - later you will see a JPA implementation)

9.3.1 Selecting pattern sources

Let's add a database source (which for now will actually be an in-memory map, until we add persistence)

- in `TutorialI18NModule`, define two pattern sources - class and in-memory (previously we were using the default - class only). The order they are declared is significant, as that is also the order they queried.

```
@Override
protected void define() {
defaultLocale(Locale.UK);
supportedLocales(Locale.GERMANY);
source(InMemory.class);
source(ClassPatternSource.class);

}
`
- The`DefaultBindingManager.addPersistenceModules()`defines a default, in-memory␣
↪store with a PatternDao implementation - no changes are therefore needed␣
↪to`BindingManager``` to include this.

If you were to run the application now, nothing will have changed. We have set the␣
↪order of bundle sources so that "in-memory store" is queried first - of course␣
↪nothing will be found as it is empty - and the "class", which will return the same␣
↪as before.

To prove this works, we need to put a value in to the in-memory store:

* in 'MyNews' add `PatternSource` and a provider for `PatternDao`. Note the␣
↪*@InMemory* annotation on `PatternDao`.
```

(continues on next page)

(continued from previous page)

```
We do not generally need to access the `PatternDao` directly, except putting values␣
↪into store - the Krail core takes care of reading patterns from the sources you␣
↪have defined in the `KrailI18NModule`
`
@Inject
protected MyNews(Translate translate, Option option, OptionPopup optionPopup,␣
↪SubjectProvider subjectProvider, UserNotifier userNotifier, @InMemory
        Provider&lt;PatternDao&gt; patternDaoProvider, PatternSource patternSource) {
    super(translate);
    nameKey=TutorialLabelKey.My_News;
    this.option = option;
    this.optionPopup = optionPopup;
    this.subjectProvider = subjectProvider;
    this.userNotifier = userNotifier;
    this.patternDaoProvider = patternDaoProvider;
    this.patternSource = patternSource;
}
`
<div class="admonition note">
<p class="first admonition-title">Note</p>
<p class="last">We find that injecting a Dao provider (as opposed to a Dao directly)␣
↪removes potential issues with persistence sessions, and recommend it as standard␣
↪practice</p>
</div>

* provide a way to enter a value for one key
** `in MyNews.doBuild()` add the code below
[source]
----
        i18NTextBox = new TextField();
        i18NTextBox.setCaption("enter a value for LabelKey.is_selected");
        submitButton = new Button("submit");
        PatternCacheKey cacheKeyUK = new PatternCacheKey(TutorialLabelKey.is_selected,
↪ Locale.UK);
        submitButton.addClickListener(event -> {
            patternSource.clearCache();
            patternDaoProvider.get().write(cacheKeyUK, i18NTextBox.getValue());
            populateBanner();
        });
        FormLayout formLayout = new FormLayout(i18NTextBox, submitButton);
        setTopRight(formLayout);

----

* change the entry for the banner to use only the first two columns (so that we can␣
↪use the top right cell)
[source]
----
   getGridLayout().addComponent(bannerLabel, 0, 0, 1, 0);
----

This provides a `TextField` to capture some input, and a submit button to submit the␣
↪value to the in memory store and update the banner. The `PatternSource` is only␣
↪needed to clear the cache (to ensure we capture the new value).

* Run the application, login and navigate to 'MyNews'
```

(continues on next page)

(continued from previous page)

```
* Make sure that the CEO New Channel is selected (we defined an I18N value for this)
* Enter some text in the "enter a value for LabelKey.is_selected", and press 'submit'
* The banner will update immediately with the text you entered
* change the Locale selector to "Deutsch" and note that the German translation is␣
↪still used - we only set a value for Locale.UK

You may recall that we defined the bundle sources like this, and noted that the␣
↪declaration order of sources is important:
```

@Override protected void define() { defaultLocale(Locale.UK); supportedLocales(Locale.GERMANY); source(InMemory.class); source(ClassPatternSource.class);

}

```
This means that the **@InMemory** source is checked first for a value - if there is␣
↪one, it is used, and the **ClassPatternSource** is not queried.  We just created a␣
↪value in the in-memory store, so that is the one that is used -this demonstrates is␣
↪why the order of declaration is important.

If you refer to the Javadoc for ```I18NModule``` (which ```KrailI18NModule```␣
↪inherits) you will see that there are methods which enable very specific settings␣
↪for the order of sources.  We will not cover that in this Tutorial, but leave you␣
↪to experiment.

#Changing Krail Core values

We have just demonstrated changing the value for a specific key - exactly the same␣
↪technique can be used to change (or add new languages to) Krail core ```I18NKey```s.
↪  This does require exporting the keys to a bundle source with mutable values␣
↪(probably a database).  The ```PatternUtility``` class provides methods to support␣
↪that process.

#Methods of configuration revisited

Earlier [in this section](tutorial-i18n-components-validation.md#config-methods) we␣
↪elected to sub-class ```KrailI18NModule``` as a way of configuring it, resulting in␣
↪this ```define()``` method:
```

@Override protected void define() { defaultLocale(Locale.UK); supportedLocales(Locale.GERMANY); source(InMemory.class); source(ClassPatternSource.class); } ‘ with this BindingManager entry ‘ @Override protected Module i18NModule() { return new TutorialI18NModule(); }

```
Because the `KrailI18NModule` methods used are all fluent, we could achieve exactly␣
↪the same by just changing the `BindingManager` like this:
`
@Override
protected Module i18NModule() {
    return new KrailI18NModule().defaultLocale(Locale.UK)
                            .supportedLocales(Locale.GERMANY)
                            .source(InMemory.class)
                            .source(ClassPatternSource.class);
}
`

The choice is yours!

= Summary
```

(continues on next page)

(continued from previous page)

```
There is still more to cover under the "I18N" heading, so the next section will cover␣
↪more of how to use Krail's I18N with Vaadin components. In this section we have:

* used `Translate` to translate an `I18NKey` directly
* translated a message with parameters
* created a *@Caption* annotation for use with your own `I18NKey`s
* added support for an additional language
* been introduced to the `CurrentLocale` class
* seen how to respond to a change of Locale message from the Event Bus
* set up a new bundle source, and determined the order of querying sources
* cleared the pattern cache
* configured Guice modules fluently and directly

= Download from GitHub

To get to this point straight from GitHub, https://github.com/davidsowerby/krail-
↪tutorial[clone] using branch *step08*

To get to this point straight from GitHub:

[source,bash]
----
git clone https://github.com/davidsowerby/krail-tutorial.git
cd krail-tutorial
git checkout --track origin/krail_0.10.0.0

----

Revert to commit _I18N Complete_
```

CHAPTER 10

Components and Validation

The previous section provided an extensive description of Krail's I18N mechanism, and gave an example of using annotations to manage the captions of a Vaadin component. This section addresses the use of I18N annotations with Vaadin Components in more detail.

10.1 Preparation

10.1.1 Set up a page

We will build a new page:

- in `MyOtherPages` add a new page entry ' addEntry("i18n", I18NDemoView.class, TutorialLabelKey.I18N, PageAccessControl.PUBLIC); '
- in package 'com.example.tutorial.pages', create a new class 'I18NDemoView' extended from `ViewBase`
- implement the `doBuild()` method
- create the enum constant *TutorialLabelKey.I18N*

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;

public class I18NDemoView extends ViewBase {

    @Inject
    protected I18NDemoView(Translate translate) {
        super(translate);
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {

    }
}
```

10.1.2 Translations

- add the following translations to `TutorialLabels_de`, creating keys where necessary

```
put(News, "Nachrichten");
put(Last_Name, "Nachname");
put(First_Name, "Vorname");
put(No, "Nein");
put(Yes, "Ja");
```

- in the 'com.example.tutorial.i18n' package, create the 'TutorialDescriptions' class

```
package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.EnumResourceBundle;

import static com.example.tutorial.i18n.TutorialDescriptionKey.*;

public class TutorialDescriptions extends EnumResourceBundle<TutorialDescriptionKey> {
    @Override
    protected void loadMap() {
        put(Interesting_Things, "Interesting things that have happened in the world.
↪");
        put(Yes,"Press for Yes");
        put(No, "Press for No");
    }
}
```

- also create the Descriptions_de class

```
package com.example.tutorial.i18n;

import static com.example.tutorial.i18n.TutorialDescriptionKey.*;

public class TutorialDescriptions_de extends TutorialDescriptions {
 @Override
 protected void loadMap() {
 put(Interesting_Things, "Interessante Dinge, die in der Welt haben geschehen");
 put(You_just_asked_for_a_pay_increase, "Sie haben für eine Lohnerhöhung gebeten");
 put(Yes, "Drücken Sie für Ja");
 put(No, "Drücken Sie für Nein");
 }
}

[source]
----

#Add different component types
```

(continues on next page)

(continued from previous page)

```
The mix of components we will use should cover all the situations you will encounter -
↪ many of the components are treated the same way for I18N, so we do not need to use␣
↪every available component.

----

package com.example.tutorial.pages;

import com.google.inject.Inject;
import com.vaadin.ui.*;
import uk.q3c.krail.core.view.ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;

public class I18NDemoView extends ViewBase {
 private Grid grid;
 private Label label;
 private Table table;
 private TextField textField;

[source]
----
@Inject
protected I18NDemoView(Translate translate) {
    super(translate);
}

@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    textField = new TextField();
    label = new Label();
    table = new Table();
    grid = new Grid();
    VerticalLayout layout = new VerticalLayout(textField, label, table, grid);
    Panel panel = new Panel();
    panel.setContent(layout);
    setRootComponent(panel);
}
----

}

[source]
----
<div class="admonition note">
<p class="first admonition-title">Note</p>
<p class="last">When you sub-class from ViewBase, make sure you set the root␣
↪component in your doBuild() method</p>
</div>

- Add the same **@TutorialCaption** to each field:
----

@TutorialCaption(caption = TutorialLabelKey.News, description =␣
↪TutorialDescriptionKey.Interesting_Things)
`
- The result should be
```

(continues on next page)

(continued from previous page)

```
`
package com.example.tutorial.pages;

import com.example.tutorial.i18n.TutorialCaption;
import com.example.tutorial.i18n.TutorialDescriptionKey;
import com.example.tutorial.i18n.TutorialLabelKey;
import com.google.inject.Inject;
import com.vaadin.ui.*;
import uk.q3c.krail.core.view.ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;

public class I18NDemoView extends ViewBase {
 @TutorialCaption(caption = TutorialLabelKey.News, description =␣
↪TutorialDescriptionKey.Interesting_Things)
 private Grid grid;
 @TutorialCaption(caption = TutorialLabelKey.News, description =␣
↪TutorialDescriptionKey.Interesting_Things)
 private Label label;
 @TutorialCaption(caption = TutorialLabelKey.News, description =␣
↪TutorialDescriptionKey.Interesting_Things)
 private Table table;
 @TutorialCaption(caption = TutorialLabelKey.News, description =␣
↪TutorialDescriptionKey.Interesting_Things)
 private TextField textField;

[source]
----
@Inject
protected I18NDemoView(Translate translate) {
    super(translate);
}

@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    textField = new TextField();
    label = new Label();
    table = new Table();
    grid = new Grid();
    VerticalLayout layout = new VerticalLayout(textField, label, table, grid);
    Panel panel = new Panel();
    panel.setContent(layout);
    setRootComponent(panel);
}
----

}

[source]
----

- Run the application and go to the 'I18N' page
- All 4 components will be present, each with a caption of 'News' and a tooltip of
↪'Interesting things that have happened in the world.'
- Changing Locale with the Locale Selector changes the language
- but only the ```TextField``` looks complete
```

(continues on next page)

(continued from previous page)

```
##Labels

Often with ```Label``` components you want to set the value of the component␣
↪statically, which you can also do with an annotation.  Actually you can do that␣
↪using Krail's I18N mechanism for any component which implements the ```com.vaadin.
↪data.Property``` interface and accepts a ```String``` value.

We have a choice to make now.  Remember that:

1. The name of an I18N annotation does not matter, it just needs to be annotated with␣
↪```@I18NAnnotation```
1. The ```I18NAnnotationProcessor``` can handle multiple annotations on the same␣
↪component
1. The annotation methods can be any combination of ```caption()```,␣
↪```description()```, ```value()``` or ```locale()```
1. We need to specify which ```I18NKey``` we use (that is, the enum class - Java will␣
↪not allow an interface as a type)

We could:

1. Add the value() method to **@Caption**
1. We could create a **@Value** annotation with only the ```value()``` method
1. We could create a caption specifically for Labels

... and quite few more choices, too.  Remember, though, that you cannot specify a␣
↪default value of **null** in an annotation, so if you want to have an annotation␣
↪method that is often not used, the best way is to specify a "null key", which␣
↪should probably return an empty ```String``` from ```Translate```

----

TutorialDescriptionKey value() default TutorialDescriptionKey.NULLKEY;
```

For the Tutorial, we will create a **@TutorialValue** annotation, which has only a `value()` method.

- in the 'com.example.tutorial.i18n' package create a new annotation 'Value'
- we will use `TutorialDescriptionKey` for values, as they can be quite long

```
package com.example.tutorial.i18n;

import uk.q3c.krail.i18n.I18NAnnotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE})
@I18NAnnotation
public @interface TutorialValue {

    TutorialDescriptionKey value();
}
```

- Add a @TutorialValue `Annotation` to the `Label`

```
@TutorialCaption(caption = TutorialLabelKey.News, description =␣
↪TutorialDescriptionKey.Interesting_Things)
@TutorialValue(value = TutorialDescriptionKey.You_just_asked_for_a_pay_increase)
private Label label;
[source]
----
- Run the application and go to the 'I18N' page
    - The ```Label``` now has a value.  Actually, we could have done the same with␣
↪the ```TextField```, but that isn't usually what you would want.
    - Change the locale with the Locale Selector, and all the captions, tooltips &␣
↪label value will change language

##Table

A ```Table``` has column headers which may need translation. If a ```Table```␣
↪propertyId is an ```I18NKey``` it will be translated - otherwise it is ignored by␣
↪the Krail ```I18NProcessor```.

- add a 'setupTable' method to ```I18NDemoView```

----

private void setupTable() {
 table.addContainerProperty(TutorialLabelKey.First_Name, String.class, null);
 table.addContainerProperty(TutorialLabelKey.Last_Name, String.class, null);
 table.setHeight("100px");
 table.setWidth("200px");
}
```

10.1.3 Grid

In a very similar way to Table, Grid may need column headings translated. If a `Grid` propertyId is an `I18NKey` it will be translated - otherwise it is ignored by the Krail `I18NProcessor`.

- add a 'setupGrid()' method

```
private void setupGrid(){
    grid.addColumn(TutorialLabelKey.First_Name, String.class);
    grid.addColumn(TutorialLabelKey.Last_Name, Integer.class);
}
```

- call these setup methods from `doBuild()`

```
@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    textField = new TextField();
    label = new Label();
    table = new Table();
    grid = new Grid();
    setupTable();
    setupGrid();
    VerticalLayout layout = new VerticalLayout(textField, label, table, grid);
    Panel panel = new Panel();
    panel.setContent(layout);
    setRootComponent(panel);
```

(continues on next page)

(continued from previous page)

```
}
```

- Run the application and go to the I18N page
  - the Table and grid now have column headings
  - Change the locale with the Locale Selector, and all the captions, tooltips, column headings & label value will change language

10.2 Drilldown and Override

There is another scenario that Krail's I18N processing supports. Assume you have a class which contains components with I18N annotations and you want to make it re-usable. Let's see how that would work.

- in the 'com.example.tutorial.i18n' package, create a new class 'ButtonBar', with **@TutorialCaption** on the buttons
- annotate the class with **@I18N** - this tells the `I18NProcessor` to drill down into this class to look for more I18N annotations. This annotation can be applied to a field or a class, but for a re-usable component it makes more sense to put it on the class.

```
package com.example.tutorial.i18n;

import com.vaadin.ui.Button;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Panel;
import uk.q3c.krail.core.i18n.I18N;

@TutorialCaption(caption = TutorialLabelKey.News, description =␣
↪TutorialDescriptionKey.Interesting_Things)
@I18N
public class ButtonBar extends Panel {

    @TutorialCaption(caption = TutorialLabelKey.Yes, description =␣
↪TutorialDescriptionKey.Yes)
    private Button yesButton;
    @TutorialCaption(caption = TutorialLabelKey.No, description =␣
↪TutorialDescriptionKey.No)
    private Button noButton;

    public ButtonBar() {
        yesButton = new Button();
        noButton = new Button();
        HorizontalLayout layout = new HorizontalLayout(yesButton,noButton);
        this.setContent(layout);
    }
}
```

- add two instances of this class to our `I18NDemoView.doBuild()`. Note that even when they are not directly nnotated, these still need to be fields (and not local variables) for the `I18NProcessor` to find the class annotations.
- include them in the layout

```
@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
textField = new TextField();
label = new Label();
table = new Table();
grid = new Grid();
buttonBar1 = new ButtonBar();
buttonBar2 = new ButtonBar();
setupTable();
setupGrid();
VerticalLayout layout = new VerticalLayout(buttonBar1,buttonBar2, textField, label,␣
↪table, grid);
Panel panel = new Panel();
panel.setContent(layout);
setRootComponent(panel);
}
[source]
----
- on the buttonBar1 field, annotate with a different **@TutorialCaption**

----

@TutorialCaption(caption = TutorialLabelKey.CEO_News_Channel,description =␣
↪TutorialDescriptionKey.Interesting_Things)
private ButtonBar buttonBar1;
```

- Run the application and the two button bars will be at the top of the page
- button bar 1 displays the caption you set at field level (overriding the class annotations)
- button bar 2 displays the caption set at class level

You could also override the drilldown specified by the ButtonBar class, simply by annotating the field with **@I18N**(drilldown=false) - although we cannot think why you might want to do that !

10.3 Form

Vaadin replaced its original Form with a `BeanFieldGroup`, which is essentially a form without the layout. Krail replaces that with its own `BeanFieldGroupBase`, which also provides integration with Krail's I18N.

To demonstrate this we need to create an entity.

- create a new package 'com.example.tutorial.form'
- in this new package create a class 'Person', and include some familiar javax validation annotations, **@Min** and **@Size**

```
package com.example.tutorial.form;

import uk.q3c.krail.persist.KrailEntity;

import javax.persistence.Id;
import javax.persistence.Version;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.Size;
```

(continues on next page)

(continued from previous page)

```
public class Person implements KrailEntity<Long,Integer> {

    @Min(0) @Max(150)
    private int age;
    @Size(min = 3)
    private String firstName;
    @Id
    private Long id;

    @Size(min=3)
    private String lastName;
    @Version
    private Integer version;

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

    public String getFirstName() {
        return firstName;
    }

    @Override
    public Long getId() {
        return id;
    }

    @Override
    public Integer getVersion() {
        return version;
    }

    public String getLastName() {
        return lastName;
    }
}
```

- Modify *build.gradle* to include javax.persistence - we have not yet introduced persistence, but we need the API for the entity
- Depending on the IDE you are using, you may need to refresh Gradle

```
dependencies {
    // remember to update the Vaadin version below if this version is changed
    compile(group: 'uk.q3c.krail', name: 'krail', version: '0.10.0.0')
```

(continues on next page)

(continued from previous page)

```
    compile 'javax.persistence:persistence-api:1.0.2'
}
```

- in package 'com.example.tutorial.form', create 'PersonForm' and create the enum constatns as required

```
package com.example.tutorial.form;

import com.example.tutorial.i18n.TutorialCaption;
import com.example.tutorial.i18n.TutorialDescriptionKey;
import com.google.inject.Inject;
import com.google.inject.Provider;
import com.vaadin.data.Property;
import com.vaadin.ui.Button;
import com.vaadin.ui.Panel;
import com.vaadin.ui.TextField;
import com.vaadin.ui.VerticalLayout;
import com.vaadin.ui.themes.ValoTheme;
import uk.q3c.krail.core.i18n.I18N;
import uk.q3c.krail.core.i18n.I18NProcessor;
import uk.q3c.krail.option.Option;
import uk.q3c.krail.core.ui.form.BeanFieldGroupBase;
import uk.q3c.krail.core.validation.BeanValidator;

import static com.example.tutorial.i18n.TutorialLabelKey.*;

@I18N
public class PersonForm extends BeanFieldGroupBase<Person> {
    @TutorialCaption(caption = Submit, description = TutorialDescriptionKey.Submit)
    private final Button submitButton;
    private final Person person;
    @TutorialCaption(caption = First_Name, description = TutorialDescriptionKey.Enter_
↪your_first_name)
    private TextField firstName;

    @TutorialCaption(caption = Last_Name, description = TutorialDescriptionKey.Enter_
↪your_last_name)
    private TextField lastName;
    @TutorialCaption(caption = Age, description = TutorialDescriptionKey.Age_of_the_
↪Person)
    private TextField age;
    @TutorialCaption(caption = Person_Form, description = TutorialDescriptionKey.
↪Person_Details_Form)
    private Panel layout;

    @Inject
    public PersonForm(I18NProcessor i18NProcessor, Provider<BeanValidator>␣
↪beanValidatorProvider, Option option) {
        super(i18NProcessor, beanValidatorProvider, option);
        firstName = new TextField();
        lastName = new TextField();
        age = new TextField();

        person = new Person();
        person.setAge(44);
        person.setFirstName("Mango");
        person.setLastName("Chutney");
        submitButton = new Button();
```

(continues on next page)

(continued from previous page)

```
        submitButton.addClickListener(event -> {
            try {
                this.commit();
            } catch (CommitException e) {
                e.printStackTrace();
            }
        });
        layout = new Panel(new VerticalLayout(firstName, lastName, age,␣
↪submitButton));
        layout.setStyleName(ValoTheme.PANEL_WELL);
        setBean(person);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void optionValueChanged(Property.ValueChangeEvent event) {

    }

    public Panel getLayout() {
        return this.layout;
    }
}
```

10.3.1 About the form

The class simply extends `BeanFieldGroupBase` , with the required entity type as a generic parameter - in this case, `Person`. Like its Vaadin counterpart, `BeanFieldGroupBase` does not concern itself with the presentation of data, or the layout of that presentation. That is the part we must provide.

You will recognise the fields and captions from the earlier part of this Tutorial section - they are just Vaadin components with **@TutorialCaption** annotations. However, it should be noted that the names of the components must match the field names of the entity to enable automatic transfer of data between the presentation layer and data model.

The constructor simply extends `BeanFieldGroupBase` and your IDE will probably auto-complete the necessary parameters. Don't forget the **@Inject** annotation though.

Within the constructor we simply build the presentation components, and define the submit button to invoke the commit() method, which will transfer data from the presentation layer back to the model - in this case the person bean.

Finally, the getLayout() method just enables a consumer class to identify the base component to place within a View.

There is an open ticket to provide more support for Forms.

- Now we need to use the form, by injecting it in to `I18NDemoView`

```
@Inject
protected I18NDemoView(Translate translate, PersonForm personForm) {
    super(translate);
    this.personForm = personForm;
}
```

*

and add it to the layout in `doBuild()`: ‘ VerticalLayout layout = new VerticalLayout(personForm.getLayout(), buttonBar1, buttonBar2, textField, label, table, grid); ‘

*

Run the application, and navigate to the I18N page

- The form will display at the top of the page with the values we have set
- change a value which breaks validation (for example, age = 443), and a validation message will appear
- change language with the Locale selector, and the language of the captions etc will change, including the validation message.

There is a more information about the Apache Bval validation integration in the Developer Guide

10.4 Summary

In this section we have:

- created and used I18N **@TutorialCaption** and **@TutorialValue** annotations
- seen how to manage `Table` and `Grid` column names for I18N
- created a re-usable I18N enabled component
- seen how to override a class I18N annotation
- created a form, with I18N integrated validation

10.5 Download from GitHub

To get to this point straight from GitHub:

```
git clone https://github.com/davidsowerby/krail-tutorial.git
cd krail-tutorial
git checkout --track origin/krail_0.10.0.0
```

Revert to commit *I18N Components and Validation Complete*

CHAPTER 11

Persistence - JPA

JPA support for Krail is provided by the krail-jpa library, which in turn is mostly provided by Apache Onami Persist. This was chosen in preference to guice-persist, primarily for its ability to support multiple concurrent database instances.

A useful comparison of Onami Persist and Guice Persist can be found here.

Krail assumes that one day you will want to use multiple persistence units - that may not be the case, but makes it easier if it is required. All this requires is to use an annotation to identify a persistence source, and it gives you an element of standardisation and future-proofing.

A generic Dao is provided (primarily for use in lambdas, but also there if that is just the way you prefer to work). Implementations are also provided for the Krail core - for `OptionDao` and I18N `PatternDao`

A reasonable understanding of JPA is assumed.

11.1 Example

We will

- create a page,
- configure two database connections (one HSQLDB and one Apache Derby),
- demonstrate some simple transactions
- demonstrate the use of JPAContainer to provide Tables, with two databases,
- demonstrate integration with Krail I18N and `Option`

We will not:

- Attempt to demonstrate all of the standard JPA capability - for that a JPA tutorial would be more appropriate

11.2 Prepare build

- include krail-jpa in the build, by adding it to *build.gradle* dependencies
- replace the existing javax dependency with *krail-jpa*. (The existing javax.persistence api is included in *krail-jpa*)

```
dependencies {
    compile 'uk.q3c.krail:krail:0.10.0.0'
    compile 'uk.q3c.krail:krail-jpa:0.10.0.0'
}
```

11.3 Create a Page

If you have followed the whole Tutorial, you will be an expert at this by now

- add a public page to `MyOtherPages`

```
addEntry("jpa", JpaView.class, TutorialLabelKey.JPA, PageAccessControl.PUBLIC);
```

- create JpaView in package 'com.example.tutorial.pages', extended from `ViewBase`

```
package com.example.tutorial.pages;

import com.google.inject.Inject;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;

public class JpaView extends ViewBase {

    @Inject
    protected JpaView(Translate translate) {
        super(translate);
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {

    }
}
```

- add the constant 'JPA' to `LabelKey`

11.4 Configure connections

This is one occasion where it may be more desirable to sub-class the relevant Guice module than use fluent methods. There is a lot that can be configured for a database instance, so configuration objects are used.

- create a new package 'com.example.tutorial.jpa'
- in this package create 'TutorialJpaModule' extended from JpaModule

```
package com.example.tutorial.jpa;

import uk.q3c.krail.persist.jpa.common.JpaModule;

public class TutorialJpaModule extends JpaModule {

    @Override
    protected void define() {

    }
}
```

- add two persistence units in the `define()` method ‘ @Override protected void define() { addPersistenceUnit(“derbyDb”, DerbyJpa.class, derbyConfig()); addPersistenceUnit(“hsqlDb”, HsqlJpa.class, hsqlConfig()); } ‘
- create the ‘DerbyJpa’ and ‘HsqlJpa’ annotations - these are Guice Binding Annotations (denoted by **@BindingAnnotation**), which will enable you to select which persistence unit you want to use from within the application.

```
package com.example.tutorial.jpa;

import com.google.inject.BindingAnnotation;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@BindingAnnotation
public @interface DerbyJpa {
}
```

```
package com.example.tutorial.jpa;

import com.google.inject.BindingAnnotation;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@BindingAnnotation
public @interface HsqlJpa {
}
```

- create a temporary folder for our Derby database. For this Tutorial we will just use the module constructor, though this is not a recommended approach for production!

```
public class TutorialJpaModule extends JpaModule {
    File userHome = new File(System.getProperty("user.home"));
    File tempDir = new File(userHome, "temp/krail-tutorial");

    public TutorialJpaModule() {

        try {
            FileUtils.forceMkdir(tempDir);
        } catch (IOException e) {
            e.printStackTrace();
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

- Provide a configuration object for each connection, using the `derbyConfig()` and `hsqlConfig()` methods. These are standard JPA configuration settings composed into a configuration object:

```
private DefaultJpaInstanceConfiguration derbyConfig() {
    DefaultJpaInstanceConfiguration config = new DefaultJpaInstanceConfiguration();
    File dbFolder = new File(tempDir, "derbyDb");

    config.transactionType(DefaultJpaInstanceConfiguration.TransactionType.RESOURCE_
↪LOCAL)
          .db(JpaDb.DERBY_EMBEDDED)
          .autoCreate(true)
          .url(dbFolder.getAbsolutePath())
          .user("test")
          .password("test")
          .ddlGeneration(DefaultJpaInstanceConfiguration.Ddl.DROP_AND_CREATE);
    return config;
}
```

```
private DefaultJpaInstanceConfiguration hsqlConfig() {
    DefaultJpaInstanceConfiguration config = new DefaultJpaInstanceConfiguration();
    config.db(JpaDb.HSQLDB)
          .autoCreate(true)
          .url("mem:test")
          .user("sa")
          .password("")
          .ddlGeneration(DefaultJpaInstanceConfiguration.Ddl.DROP_AND_CREATE);
    return config;
}
```

- update the `BindingManager` to make it aware of this new module. This would override the use of the default `InMemoryModule`, but we want that as well for demonstration purposes ‘ @Override protected void addPersistenceModules(List<Module> modules) { super.addPersistenceModules(modules); modules.add(new TutorialJpaModule()); } ‘
- Unfortunately we still need a minimal persistence.xml file, so we need to
  - create folder src/main/resources/META-INF
  - create the following *persistence.xml* file in that folder

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.
↪sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="derbyDb">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
        </properties>

    </persistence-unit>

    <persistence-unit name="hsqlDb">
```

(continues on next page)

(continued from previous page)

```
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
        </properties>

    </persistence-unit>
</persistence>
```

11.5 Prepare the service

- configure the `TutorialServletModule` to add the `PersistenceFilter`

```
package com.example.tutorial.app;

import org.apache.onami.persist.PersistenceFilter;
import uk.q3c.krail.core.guice.BaseServletModule;

public class TutorialServletModule extends BaseServletModule {

    @Override
    protected void configureServlets() {
        filter("/*").through(PersistenceFilter.class);
        serve("/*").with(TutorialServlet.class);
    }
}
```

11.6 Prepare the Entity

- Update the `Person` entity we used earlier, to be JPA compliant
  - add the **@Entity** class annotation
  - use auto-generated id

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

11.7 Prepare the user interface

- set up the basic layout components in `JpaView` ‘ @Override protected void doBuild(ViewChangeBusMessage busMessage) { Panel panel = new Panel(); setRootComponent(panel); } ‘

In `JpaView` we want to show a table each for the Derby and HSQLDB connections. A Vaadin `Table` uses a `Container` to provide the data, and in this case a `JPAContainer`.

- To get a container, we need to inject a `JpaContainerProvider` for each persistence unit, identified by their annotations, **@DerbyJpa** and **@HsqlJpa**

```
package com.example.tutorial.pages;

import com.example.tutorial.jpa.DerbyJpa;
import com.example.tutorial.jpa.HsqlJpa;
import com.google.inject.Inject;
import com.vaadin.ui.Panel;
import uk.q3c.krail.core.option.jpa.JpaContainerProvider;
import uk.q3c.krail.core.view.ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;

public class JpaView extends ViewBase {

    private JpaContainerProvider derbyContainerProvider;
    private JpaContainerProvider hsqlContainerProvider;

    @Inject
    protected JpaView(Translate translate, @DerbyJpa JpaContainerProvider␣
↪derbyContainerProvider, @HsqlJpa JpaContainerProvider hsqlContainerProvider) {
        super(translate);
        this.derbyContainerProvider = derbyContainerProvider;
        this.hsqlContainerProvider = hsqlContainerProvider;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        Panel panel = new Panel();
        setRootComponent(panel);
    }
}
```

- completing the layout so that the JPA data is presented in Vaadin Tables, via JPAContainers, `JpaView` should be like this:

```
package com.example.tutorial.pages;

import com.example.tutorial.form.Person;
import com.example.tutorial.jpa.DerbyJpa;
import com.example.tutorial.jpa.HsqlJpa;
import com.google.inject.Inject;
import com.vaadin.addon.jpacontainer.JPAContainer;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Panel;
import com.vaadin.ui.Table;
import com.vaadin.ui.VerticalLayout;
import uk.q3c.krail.core.option.jpa.JpaContainerProvider;
import uk.q3c.krail.core.view.ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.persist.ContainerType;

public class JpaView extends ViewBase {

    private JpaContainerProvider derbyContainerProvider;
    private JpaContainerProvider hsqlContainerProvider;
    private JPAContainer<Person> derbyContainer;
    private JPAContainer<Person> hsqlContainer;
```

(continues on next page)

(continued from previous page)

```
    private Table derbyTable;
    private Table hsqlTable;

    @Inject
    protected JpaView(Translate translate, @DerbyJpa JpaContainerProvider␣
↪derbyContainerProvider, @HsqlJpa JpaContainerProvider hsqlContainerProvider) {
        super(translate);
        this.derbyContainerProvider = derbyContainerProvider;
        this.hsqlContainerProvider = hsqlContainerProvider;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        derbyContainer=derbyContainerProvider.get(Person.class, ContainerType.CACHED);
        hsqlContainer=hsqlContainerProvider.get( Person.class, ContainerType.CACHED);
        derbyTable = new Table("",derbyContainer);
        hsqlTable = new Table("",hsqlContainer);

        VerticalLayout derbyLayout = new VerticalLayout(derbyTable);
        VerticalLayout hsqlLayout = new VerticalLayout(hsqlTable);

        HorizontalLayout horizontalLayout=new HorizontalLayout(derbyLayout,
↪hsqlLayout);
        Panel panel = new Panel();
        panel.setContent(horizontalLayout);
        setRootComponent(panel);
    }
}
```

The Vaadin ‘Table‘s, are using containers from the ‘JpaContainerProvider‘s to provide the data

- Now we need to provide the I18N captions for the `Table` components

```
@TutorialCaption(caption = TutorialLabelKey.Derby_Table, description =␣
↪TutorialDescriptionKey.Table_connected_to_DerbyDb)
private Table derbyTable;
@TutorialCaption(caption = TutorialLabelKey.HSQL_Table, description =␣
↪TutorialDescriptionKey.Table_connected_to_HsqlDb)
private Table hsqlTable;
```

- run the application just to make sure you have everything correctly set up so far. There is no data to display yet, so all you will see is two empty tables.

11.8 Data

- in `JPAView`, create a convenience method for creating new people. This is so much quicker than the conventional method for creating people, but nowhere near as much fun. ‘ private Person createPerson() { Person p = new Person(); int i=new Random().nextInt(5000); p.setAge(i % 80); p.setFirstName(“First name “+i); p.setLastName(“Last name ” + i); return p; } ‘

There are different ways of accessing the data.

11.8.1 Using the EntityManager

This is the method recommended by the Apache Onami team:

- inject an `EntityManagerProvider` (The Onami provider, not the Vaadin provider) for each persistence unit, using the binding annotations to identify them

```
@Inject
protected JpaView(Translate translate, @DerbyJpa JpaContainerProvider␣
↪derbyContainerProvider, @HsqlJpa JpaContainerProvider hsqlContainerProvider,
↪@DerbyJpa EntityManagerProvider derbyEntityManagerProvider, @HsqlJpa␣
↪EntityManagerProvider hsqlEntityManagerProvider) {
    super(translate);
    this.derbyContainerProvider = derbyContainerProvider;
    this.hsqlContainerProvider = hsqlContainerProvider;
    this.derbyEntityManagerProvider = derbyEntityManagerProvider;
    this.hsqlEntityManagerProvider = hsqlEntityManagerProvider;
}
```

- create a method to undertake the transaction

```
@Transactional
protected void addWithEntityMgr(EntityManagerProvider entityManagerProvider) {
    final EntityManager entityManager = entityManagerProvider.get();
    entityManager.persist(createPerson());
}
```

- add two buttons to call the `addWithEntityMgr` method, and refresh the container (so that we can see the changes)
- add the buttons to the vertical layouts. The complete `doBuild()` method now looks like this:

```
@Override
protected void doBuild(ViewChangeBusMessage busMessage) {
    derbyContainer=derbyContainerProvider.get(Person.class, ContainerType.CACHED);
    hsqlContainer=hsqlContainerProvider.get( Person.class, ContainerType.CACHED);
    derbyTable = new Table("",derbyContainer);
    hsqlTable = new Table("",hsqlContainer);

    derbyEntityMgrButton = new Button();
    derbyEntityMgrButton.addClickListener(event -> {
        addWithEntityMgr(derbyEntityManagerProvider);
        derbyContainer.refresh();
    });
    hsqlEntityMgrButton = new Button();
    hsqlEntityMgrButton.addClickListener(event -> {
        addWithEntityMgr(hsqlEntityManagerProvider);
        hsqlContainer.refresh();
    });

    VerticalLayout derbyLayout = new VerticalLayout(derbyTable, derbyEntityMgrButton);
    VerticalLayout hsqlLayout = new VerticalLayout(hsqlTable, hsqlEntityMgrButton);

    HorizontalLayout horizontalLayout=new HorizontalLayout(derbyLayout,hsqlLayout);
    Panel panel = new Panel();
    panel.setContent(horizontalLayout);
    setRootComponent(panel);
}
```

- give the buttons captions and descriptions

```
@TutorialCaption(caption = TutorialLabelKey.Add_with_entity_manager, description =␣
↪TutorialDescriptionKey.Add_with_entity_manager)
private Button derbyEntityMgrButton;
@TutorialCaption(caption = TutorialLabelKey.Add_with_entity_manager, description =␣
↪TutorialDescriptionKey.Add_with_entity_manager)
private Button hsqlEntityMgrButton;
```

- run the application and press the buttons
  - you will see that each persistence unit is operating separately, just by use of the binding annotations

11.8.2 DAO

There is a lot of debate about the value of using DAOs; we generally only use them where there is a particular value in doing so. One such case, we believe, is where you are using a lot of Java 8 lambdas to respond, for example, to button clicks. JPA would require a separate, annotated method for each type of response needed.

For this use case Krail provides a generic DAO for the simple JPA calls to avoid the need for creating those annotated methods.

- inject the DAO for each persistence unit

```
@Inject
protected JpaView(Translate translate, @DerbyJpa JpaContainerProvider␣
↪derbyContainerProvider, @HsqlJpa JpaContainerProvider hsqlContainerProvider,
                  @DerbyJpa EntityManagerProvider derbyEntityManagerProvider,␣
↪@HsqlJpa EntityManagerProvider hsqlEntityManagerProvider, @DerbyJpa JpaDao_LongInt␣
↪derbyDao, @HsqlJpa JpaDao_LongInt hsqlDao) {
    super(translate);
    this.derbyContainerProvider = derbyContainerProvider;
    this.hsqlContainerProvider = hsqlContainerProvider;
    this.derbyEntityManagerProvider = derbyEntityManagerProvider;
    this.hsqlEntityManagerProvider = hsqlEntityManagerProvider;
    this.derbyDao = derbyDao;
    this.hsqlDao = hsqlDao;
}
```

- DAOs are not bound automatically, so we add them to the persistence unit configuration in `TutorialJpaModule` by calling `useLongIntDao()` on the `JpaInstanceConfiguration` (on both configs)

```
private DefaultJpaInstanceConfiguration derbyConfig() {
    DefaultJpaInstanceConfiguration config = new DefaultJpaInstanceConfiguration();
    File dbFolder = new File(tempDir, "derbyDb");

    config.transactionType(DefaultJpaInstanceConfiguration.TransactionType.RESOURCE_
↪LOCAL)
          .db(JpaDb.DERBY_EMBEDDED)
          .autoCreate(true)
          .url(dbFolder.getAbsolutePath())
          .useLongIntDao()
          .user("test")
          .password("test")
          .ddlGeneration(DefaultJpaInstanceConfiguration.Ddl.DROP_AND_CREATE);
    return config;
```

(continues on next page)

(continued from previous page)

```
}

private DefaultJpaInstanceConfiguration hsqlConfig() {
    DefaultJpaInstanceConfiguration config = new DefaultJpaInstanceConfiguration();
    config.db(JpaDb.HSQLDB)
          .autoCreate(true)
          .url("mem:test")
          .useLongIntDao()
          .user("sa")
          .password("")
          .ddlGeneration(DefaultJpaInstanceConfiguration.Ddl.DROP_AND_CREATE);
    return config;
}
```

- add buttons to `JpaView.doBuild()`

```
//add with Dao
derbyDaoButton = new Button();
derbyDaoButton.addClickListener(event -> {
    derbyDao.save(createPerson());
    derbyContainer.refresh();
});
hsqlDaoButton = new Button();
hsqlDaoButton.addClickListener(event -> {
    hsqlDao.save(createPerson());
    hsqlContainer.refresh();
});
```

- include them in the layout ‘ VerticalLayout derbyLayout = new VerticalLayout(derbyTable, derbyEntityMgrButton, derbyDaoButton); VerticalLayout hsqlLayout = new VerticalLayout(hsqlTable, hsqlEntityMgrButton, hsqlDaoButton); ‘
- give them I18N captions and descriptions

```
@TutorialCaption(caption = TutorialLabelKey.Add_with_DAO, description =␣
↪TutorialDescriptionKey.Add_with_DAO)
private Button derbyDaoButton;
@TutorialCaption(caption = TutorialLabelKey.Add_with_DAO, description =␣
↪TutorialDescriptionKey.Add_with_DAO)
private Button hsqlDaoButton;
```

- run the application, navigate to JPA
  - the “add with DAO” buttons work in the same way as the “add with EntityManager” buttons

11.9 Persistence for Option

addEntry(“jpa/option”, JpaOptionView.class, LabelKey.Options, PageAccessControl.PUBLIC);

```
- create a new class JpaOptionView in the 'pages' package

[source]
----
```

(continues on next page)

(continued from previous page)

```
package com.example.tutorial.pages;

import com.example.tutorial.i18n.Caption;
import com.example.tutorial.i18n.DescriptionKey;
import com.example.tutorial.i18n.LabelKey;
import com.example.tutorial.jpa.DerbyJpa;
import com.google.inject.Inject;
import com.vaadin.addon.jpacontainer.JPAContainer;
import com.vaadin.data.Property;
import com.vaadin.ui.Button;
import com.vaadin.ui.HorizontalLayout;
import com.vaadin.ui.Panel;
import com.vaadin.ui.Table;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.option.*;
import uk.q3c.krail.persist.ContainerType;
import uk.q3c.krail.persist.VaadinContainerProvider;
import OptionEntity;
import uk.q3c.krail.core.persist.inmemory.InMemoryContainer;
import uk.q3c.krail.core.view.ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.persist.jpa.common.JpaContainerProvider;
import uk.q3c.krail.persist.jpa.option.JpaOptionEntity;

import javax.annotation.Nonnull;

public class JpaOptionView extends ViewBase implements OptionContext {

    public static final OptionKey<String> anyOldText = new OptionKey<>("default text",
↪ MyNews.class, LabelKey.Age, DescriptionKey.Age_of_the_Person);
    private final VaadinContainerProvider inMemoryContainerProvider;
    private final JpaContainerProvider derbyContainerProvider;
    private JPAContainer<JpaOptionEntity> derbyContainer;
    private InMemoryContainer inMemoryContainer;

    @Caption(caption = LabelKey.In_Memory, description = DescriptionKey.Interesting_
↪Things )
    private Table inMemoryTable;
    @Caption(caption = LabelKey.Derby, description = DescriptionKey.Interesting_
↪Things )
    private Table derbyTable;
    private Option option;
    private OptionPopup optionPopup;

    @Caption(caption = LabelKey.Options, description = DescriptionKey.Interesting_
↪Things )
    private Button optionPopupButton;

    @Inject
    protected JpaOptionView(Translate translate, @InMemory VaadinContainerProvider␣
↪inMemoryContainerProvider, @DerbyJpa JpaContainerProvider
            derbyContainerProvider, OptionPopup
            optionPopup, Option option) {
        super(translate);
        this.inMemoryContainerProvider = inMemoryContainerProvider;
        this.derbyContainerProvider = derbyContainerProvider;
        this.optionPopup = optionPopup;
```

(continues on next page)

(continued from previous page)

```
        this.option = option;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        optionPopupButton = new Button();
        optionPopupButton.addClickListener(event -> optionPopup.popup(this, LabelKey.
↪Options));
        inMemoryTable = new Table();
        derbyTable = new Table();
        inMemoryContainer = (InMemoryContainer) inMemoryContainerProvider.
↪get(OptionEntity.class, ContainerType.CACHED);
        derbyContainer = derbyContainerProvider.get(JpaOptionEntity.class,␣
↪ContainerType.CACHED);
        inMemoryTable.setContainerDataSource(inMemoryContainer);
        derbyTable.setContainerDataSource(derbyContainer);

        HorizontalLayout horizontalLayout = new HorizontalLayout(optionPopupButton,␣
↪inMemoryTable, derbyTable);
        setRootComponent(new Panel(horizontalLayout));

    }


    @Override
    public Option getOption() {
        return option;
    }

    @Override
    public void optionValueChanged(Property.ValueChangeEvent event) {
        inMemoryContainer.refresh();
        derbyContainer.refresh();
    }
}

----

There is quite a lot in this class, but you have seen most of it already - these are␣
↪the key points:

* an `OptionKey` is defined purely for demonstrating a change of value
* We are injecting ContainerProviders to provide Vaadin Container instances fro the␣
↪Vaadin Tables
* A Vaadin `Table` is used for each persistence source to present the data
* the `OptionPopup` is used so that we can change the value of an `Option`
* the `optionValueChanged()` method refreshes the both `Container` (and associated␣
↪`Table`) instances when an `Option` value is changed

We also need to update _persistence.xml_ to include `JpaOptionEntity`:

[source,xml]
----
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.
↪sun.com/xml/ns/persistence/persistence_2_0.xsd"
```

(continues on next page)

(continued from previous page)

```
            version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="derbyDb">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>uk.q3c.krail.persist.jpa.option.JpaOptionEntity</class>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
        </properties>

    </persistence-unit>

    <persistence-unit name="hsqlDb">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
        </properties>

    </persistence-unit>
</persistence>
----

Now to check what is happening:

* run the application and log in (for example 'eq'/'eq') so that you can change the␣
↪option value
* navigate to "JPA | Options"
* click the "Options" button and change the option value
* the "In Memory" table will update

== Changing to JPA

* configure the JPA provider to bind an `OptionDao`. This is done by amending the␣
↪config in `TutorialJpaModule` to include a call to `provideOptionDao()`:
[source]
----
   private DefaultJpaInstanceConfiguration derbyConfig() {
        DefaultJpaInstanceConfiguration config = new␣
↪DefaultJpaInstanceConfiguration();
        File dbFolder = new File(tempDir, "derbyDb");

        config.transactionType(DefaultJpaInstanceConfiguration.TransactionType.
↪RESOURCE_LOCAL)
                .db(JpaDb.DERBY_EMBEDDED)
                .autoCreate(true)
                .url(dbFolder.getAbsolutePath())
                .user("test")
                .useLongIntDao()
                .provideOptionDao()
                .password("test")
                .ddlGeneration(DefaultJpaInstanceConfiguration.Ddl.DROP_AND_CREATE);
        return config;
    }
----

* select *@DerbyJpa* as the active source for `Option` by modifying the␣
↪`BindingManager`:
[source]
----
```

(continues on next page)

(continued from previous page)

```
@Override
protected Module optionModule() {
    return new OptionModule().activeSource(DerbyJpa.class);
}
----

* Run the application and log in
* navigate to "JPA | Options"
* click the "Options" button and change the option value
* the "Derby" table will update instead of the "In Memory" table

<a name="persistence-i18n"></a>

= Persistence for I18N

Persistence for I18N patterns is a little different to persistence for `Option`. For␣
↪`Option`, there is only ever one source in use, but as we have already seen, we can␣
↪use multiple sources for I18N patterns, working in a hierarchy.

To demonstrate this we will go back to the JPA page - and if you wish to check first,␣
↪you will see that none of the Tutorial display for this page is translated.

We will simulate a real world requirement to hold translations in a database by␣
↪adding a translation to the Derby source, and then updating the configuration and␣
↪see the translation take effect.

This is also what you would do if you want to change or add translations to the Krail␣
↪core - export the patterns to a mutable source and update / add the translations.

* add the *@DerbyJpa* pattern dao to the constructor injections
`
@Inject
protected JpaView(Translate translate, @DerbyJpa JpaContainerProvider␣
↪derbyContainerProvider, @HsqlJpa JpaContainerProvider hsqlContainerProvider,
              @DerbyJpa EntityManagerProvider derbyEntityManagerProvider, @HsqlJpa␣
↪EntityManagerProvider hsqlEntityManagerProvider, @DerbyJpa
                  JpaDao_LongInt derbyDao, @HsqlJpa JpaDao_LongInt hsqlDao, @DerbyJpa␣
↪PatternDao patternDao) {
super(translate);
this.derbyContainerProvider = derbyContainerProvider;
this.hsqlContainerProvider = hsqlContainerProvider;
this.derbyEntityManagerProvider = derbyEntityManagerProvider;
this.hsqlEntityManagerProvider = hsqlEntityManagerProvider;
this.derbyDao = derbyDao;
this.hsqlDao = hsqlDao;
this.patternDao = patternDao;
}
`
* create a button to insert a new value into the Derby pattern table:
[source]
----
derbyPatternButton = new Button();
derbyPatternButton.addClickListener(event->{patternDao.write(new␣
↪PatternCacheKey(LabelKey.Derby_Table, Locale.GERMANY),"Tafel aus Derby");});

VerticalLayout derbyLayout = new VerticalLayout(derbyTable, derbyEntityMgrButton,␣
↪derbyDaoButton,derbyPatternButton);
```

(continues on next page)

(continued from previous page)

```
----

* provide a caption and description
[source]
----
@Caption(caption = LabelKey.Insert_Pattern_value, description = DescriptionKey.Insert_
↪Pattern_value)
private Button derbyPatternButton;
----

* In the same way as we did for `Option`, set up the Derby configuration in␣
↪`TutorialJpaModule` to produce a pattern dao by a call to `providePatterDao()`
[source]
----
private DefaultJpaInstanceConfiguration derbyConfig() {
    DefaultJpaInstanceConfiguration config = new DefaultJpaInstanceConfiguration();
    File dbFolder = new File(tempDir, "derbyDb");

    config.transactionType(DefaultJpaInstanceConfiguration.TransactionType.RESOURCE_
↪LOCAL)
          .db(JpaDb.DERBY_EMBEDDED)
          .autoCreate(true)
          .url(dbFolder.getAbsolutePath())
          .useLongIntDao()
          .provideOptionDao()
          .providePatternDao()
          .user("test")
          .password("test")
          .ddlGeneration(DefaultJpaInstanceConfiguration.Ddl.DROP_AND_CREATE);
    return config;
}
----

* instruct the I18NModule to use *@DerbyJpa* as a source - we will put it in first␣
↪place to ensure that it is picked up first - but we still want to use the Class␣
↪based definitions if there is nothing in the Derby source:
[source]
----
@Override
protected Module i18NModule() {
    return new TutorialI18NModule().source(DerbyJpa.class)
                                   .source(ClassPatternSource.class);
}
----

* add the JPA pattern entity to _persistence.xml_
[source,xml]
----
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.
↪sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="derbyDb">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>uk.q3c.krail.persist.jpa.option.JpaOptionEntity</class>
```

(continues on next page)

(continued from previous page)

```
        <class>uk.q3c.krail.persist.jpa.i18n.JpaPatternEntity</class>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
        </properties>

    </persistence-unit>

    <persistence-unit name="hsqlDb">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
        </properties>

    </persistence-unit>
</persistence>
----

* run the application and navigate to "JPA"
* press the "Insert Pattern Value" button to save a translation for "Derby Table"␣
↪into the *@DerbyJpa* PU
* use the Locale selector to change to "Deutsch"
* The caption for the Derby table now shows the German translation

= Summary

We have :

* configured two database connections (one HSQLDB and One Apache Derby),
* kept the In Memory source, working in conjunction with JPA sources
* demonstrated some simple transactions using method annotation
* demonstrated transactions from within a lambda
* used the generic DAO for both JPA sources
* used JPA containers, with sources identified by annotation
* configured `Option` to use JPA persistence
* configured I18N to use JPA for pattern persistence
* demonstrated the hierarchical nature of I18N pattern sources, so that there is␣
↪always a translation

= Download from GitHub

To get to this point straight from GitHub, https://github.com/davidsowerby/krail-
↪tutorial[clone] using branch *step09*
```

CHAPTER 12

Guice & Scopes

For this short section we will not be actually adding to the Tutorial application - it is time for just a little bit of theory.

12.1 Introduction

As a developer you will be familiar with the idea of scope, even if you have never used Guice before - Guice scopes are no different in principle than considering the scope, for example, of a local vs global variable. The Guice documentation is an excellent place to start for understanding Guice itself, and the section on scopes is particularly relevant.

In this section we will consider the way in which scopes are implemented by Krail.

12.2 Singleton

A Singleton has only one instance in the application. Krail uses the standard Guice Singleton with no changes. All Singletons must be thread safe.

12.3 VaadinSessionScoped

The unique environment* of a `VaadinSession` requires a custom Guice scope of **@VaadinSessionScoped** - and is generally equivalent to a browser instance. Classes of this scope should be thread safe, as a Vaadin Session may use multiple threads.

12.4 UIScoped

The Vaadin UI is generally equivalent to a browser tab, and requires a custom Guice scope of **@UIScoped**. Classes of this scope do not need to be thread safe.

12.5 Applying a scope

All of the above scopes may be applied as described here

12.6

Note that the standard web annotation of *@SessionScoped appears to work except when using with Vaadin Push - but has not been tested thoroughly.

CHAPTER 13

Event Bus

13.1 Introduction

Krail integrates the event bus provided by MBassador. For more information about the integration itself, see the this project's contribution to the MBassador documentation.

There is no point duplicating MBassador's documentation here, but in brief, MBassador enables the use of synchronous or asynchronous event (message) buses. MBassador is a sophisticated, high performance event bus, and it is strongly recommended that you read its documentation to get the best from it.

 There is a logical correlation between an event bus and a Guice scope, and that is what Krail provides - an event bus for Singleton, VaadinSession and UI scopes as described in the Guice Scopes chapter. These can be accessed by:

- annotation (**@UiBus**, **@SessionBus**, **@GlobalMessageBus**)
- provider (`UIBusProvider`, `SessionBusProvider`, `GlobalBusProvider`)

13.2 The Tutorial task

We will create 3 buttons to publish messages, and receivers for events of each scope (UI, Session and Global). By sending messages via the different buses we will be able to see how scope affects where the messages are received.

13.3 Create a page

If you have followed the Tutorial up to this point you will now be a complete expert in creating pages. However, just in case you have stepped in to the Tutorial part way through (do developers really do that?), this is what you need to do:

- Amend the `OtherPages` module by adding the following line to the `define()` method:

```
addEntry("events", EventsView.class, LabelKey.Events, PageAccessControl.PERMISSION);
```

- create the enum constant for the page
- create the view `EventsView` in *com.example.tutorial.pages* (code is provided later)
- create a package *com.example.tutorial.eventbus*
- in this new package, create a `TutorialMessage` class
- Our `TutorialMessage` will carry a String message and the sender. Copy the following:

```
package com.example.tutorial.eventbus;

import uk.q3c.krail.eventbus.BusMessage;

public class TutorialMessage implements BusMessage {

    private final String msg;
    private Object sender;

    public TutorialMessage(String msg, Object sender) {
        this.msg = msg;
        this.sender = sender;
    }

    public String getMsg() {
        return msg + " from " + Integer.toHexString(sender.hashCode());
    }
}
```

13.4 Message receivers

We will create a simple component to accept messages from a bus and display them in a `TextArea`, and use this as a base class for each message receiver.

13.4.1 Base class

- create a new class, `MessageReceiver` in *com.example.tutorial.eventbus*
- copy the code below

```
package com.example.tutorial.eventbus;

import com.vaadin.ui.Panel;
import com.vaadin.ui.TextArea;
import net.engio.mbassy.listener.Handler;

public abstract class MessageReceiver extends Panel {
    private final TextArea textField;

    public MessageReceiver() {
        this.setSizeFull();
        this.textField = new TextArea();
        textField.setSizeFull();
        textField.setRows(8);
        setContent(textField);
    }
```

(continues on next page)

(continued from previous page)

```
    public String getText() {
        return textField.getValue();
    }

    @Handler
    public void addMsg(TutorialMessage tutorialMessage) {
        String s = getText();
        textField.setValue(s+"\n"+tutorialMessage.getMsg());
    }
}
```

The **@Handler** annotation ensures the `addMsg()` method intercepts all `TutorialMessage` events which are passed by the bus(es) which the class is subscribed to. We will subscribe in the following sub-classes, so that each one intercepts `TutorialMessage` events for a specific bus - but you can subscribe to multiple buses.

13.4.2 Receiver for each bus

- create three sub-classes, `GlobalMessageReceiver`, `SessionMessageReceiver` and `UIMessageReceiver` each extending `MessageReceiver`, in *com.example.tutorial.eventbus*

```
package com.example.tutorial.eventbus;

import net.engio.mbassy.listener.Listener;
import uk.q3c.krail.eventbus.GlobalBus;
import uk.q3c.krail.eventbus.SubscribeTo;

@Listener @SubscribeTo(GlobalBus.class)
public class GlobalMessageReceiver extends MessageReceiver {
}
```

```
package com.example.tutorial.eventbus;

import net.engio.mbassy.listener.Handler;
import net.engio.mbassy.listener.Listener;
import uk.q3c.krail.core.eventbus.SessionBus;
import uk.q3c.krail.eventbus.SubscribeTo;

@Listener @SubscribeTo(SessionBus.class)
public class SessionMessageReceiver extends MessageReceiver {

}
```

```
package com.example.tutorial.eventbus;

import net.engio.mbassy.listener.Listener;
import uk.q3c.krail.eventbus.SubscribeTo;
import uk.q3c.krail.core.eventbus.UIBus;

@Listener @SubscribeTo(UIBus.class)
public class UIMessageReceiver extends MessageReceiver {
}
```

The **@Listener** annotation marks the class as an `MBassador` bus subscriber. The **@SubscribeTo** annotation is a Krail annotation to identify which bus or buses the class should be subscribed to. The **@SubscribeTo** annotation is

processed by Guice AOP, therefore the class must be instantiated by Guice for it to work.

You could achieve the same by injecting a bus and directly subscribing:

```
globalBusProvider.get().subscribe(this)
```

13.5 Completing the View

- cut and paste the code below into `EventsView`

```
package com.example.tutorial.pages;

import com.example.tutorial.eventbus.*;
import com.example.tutorial.i18n.Caption;
import com.example.tutorial.i18n.DescriptionKey;
import com.example.tutorial.i18n.LabelKey;
import com.google.inject.Inject;
import com.vaadin.ui.Button;
import uk.q3c.krail.eventbus.GlobalBusProvider;
import uk.q3c.krail.core.eventbus.SessionBusProvider;
import uk.q3c.krail.core.eventbus.UIBusProvider;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;

public class EventsView extends Grid3x3ViewBase {
    private final UIBusProvider uiBusProvider;
    private final GlobalBusProvider globalBusProvider;
    @Caption(caption = LabelKey.Singleton, description = DescriptionKey.Singleton)
    private Button singletonSendBtn;
    @Caption(caption = LabelKey.Session, description = DescriptionKey.Session)
    private Button sessionSendBtn;
    @Caption(caption = LabelKey.UI, description = DescriptionKey.UI)
    private Button uiSendBtn;
    @Caption(caption = LabelKey.Refresh, description = DescriptionKey.Refresh)
    private Button refreshBtn;
    private SessionBusProvider sessionBusProvider;
    private GlobalMessageReceiver singletonMessageReceiver;
    private MessageReceiver sessionMessageReceiver;
    private MessageReceiver uiMessageReceiver;

    @Inject
    protected EventsView(Translate translate,UIBusProvider uiBusProvider,␣
↪SessionBusProvider sessionBusProvider, GlobalBusProvider globalBusProvider,
                         GlobalMessageReceiver singletonMessageReceiver,␣
↪SessionMessageReceiver sessionMessageReceiver, UIMessageReceiver uiMessageReceiver)
↪{
        super(translate);
        this.uiBusProvider = uiBusProvider;
        this.sessionBusProvider = sessionBusProvider;
        this.singletonMessageReceiver = singletonMessageReceiver;
        this.sessionMessageReceiver = sessionMessageReceiver;
        this.uiMessageReceiver = uiMessageReceiver;
        this.globalBusProvider = globalBusProvider;
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        constructEventSendButtons();
        layoutReceivers();
        refreshBtn = new Button();
        setTopRight(refreshBtn);
    }

    private void layoutReceivers() {
        setTopCentre(singletonMessageReceiver);
        setMiddleCentre(sessionMessageReceiver);
        setBottomCentre(uiMessageReceiver);
    }

    private void constructEventSendButtons() {
        singletonSendBtn = new Button();
        sessionSendBtn = new Button();
        uiSendBtn = new Button();
        singletonSendBtn.addClickListener(click -> {
            String m = "Singleton";
            globalBusProvider.get()
                             .publish(new TutorialMessage(m,this));
        });
        sessionSendBtn.addClickListener(click -> {
            String m = "Session";
            sessionBusProvider.get()
                              .publish(new TutorialMessage(m,this));
        });
        uiSendBtn.addClickListener(click -> {
            String m = "UI";
            uiBusProvider.get()
                         .publish(new TutorialMessage(m,this));
        });
        setTopLeft(singletonSendBtn);
        setMiddleLeft(sessionSendBtn);
        setBottomLeft(uiSendBtn);
    }
}
```

- create the enum constants

The `constructEventSendButtons()` method provides a button for each bus to send a message.

A bus for each scope is injected into the constructor using BusProviders

The Refresh button appears to do nothing, but that will become clear later.

A `MessageReceiver` is injected for each bus (remember these need to be instantiated by Guice)

13.6 Demonstrating the result

- run the application
- open a browser, which we will call browser 1 tab 1
- login as *'admin'*, *'password'*

- navigate to the *Event Bus* page
- open a second browser tab at the same URL - we will call this browser 1 tab 2 (now that surprised you!)
- in browser 1 tab 1 press each of the 3 buttons, Singleton, Session and UI
- Messages will appear in all 3 text areas
- Switch to tab 2 (there will be no messages visible yet)

If you know Vaadin, you will be familiar with this situation - the Vaadin client is unaware that changes have been made on the server, so the display has not been updated. It will only update when the client is prompted to get an update from the server. (We will come back to this when we address Vaadin Push). For our purposes, we just click the Refresh button. This actually does nothing except cause the client to poll the server for updates.

- click Refresh
- the Singleton and Session text areas will contain a message from the same source, but the UI area will be empty

This demonstrates the scope of the event buses. The UI bus is of UIScope - which means it relates to a browser tab (unless embedded). The session scope relates to a browser instance, and therefore appears in both tabs, and a singleton scope applies to an application and also appears in both tabs.

- open a second browser instance (if you are using Chrome, be aware that Chrome does odd things with browser instances - to be certain you have a separate instance, it is better to use Firefox as the second instance)
- in browser 2, login as *'admin'*, *'password'*
- navigate to the *Event Bus* page
- switch back to browser 1 tab 1 and press each of the 3 buttons, Singleton, Session and UI again
- switch browser 2 tab 1
- press Refresh
- Only the Singleton text area will contain a message

This is what we expect - a Vaadin session relates to a browser instance, so a session message will not appear in browser 2 - only the Singleton will

13.7 Summary

- We have covered the 3 defined event buses provided by Krail, with Singleton, Session and UI scope
- We have seen how to subscribe to a bus
- We have seen how to publish to a bus
- We have identified a challenge with refreshing the Vaadin client

13.8 Download from GitHub

To get to this point straight from GitHub, clone using branch **step10**

CHAPTER 14

Services

The Guice documentation strongly recommends making Guice modules fast and side effect free. It also provides an example interface for starting and stopping services.

Krail extends that idea with a more comprehensive lifecycle for a Service, and also adds dependency management. For example, in order to start a Database Service, it may be necessary to load configuration values from a file or web service first.

14.1 Lifecycle

The lifecycle is defined by `Service.State` and the standard cycle comprises states:

- INITIAL, STARTING, RUNNING, STOPPING, STOPPED plus a state of FAILED

The transient states of STARTING and STOPPING are there because some services may take a while to fully start or stop.

14.1.1 Causes

package com.example.tutorial.service;

import com.example.tutorial.i18n.LabelKey; import com.google.inject.Inject; import uk.q3c.krail.eventbus.GlobalBusProvider; import uk.q3c.krail.service.AbstractService; import uk.q3c.krail.service.ServiceModel; import uk.q3c.krail.i18n.I18NKey; import uk.q3c.krail.i18n.Translate;

@Singleton public class ServiceA extends AbstractService {

```
@Inject
protected ServiceA(Translate translate, ServicesModel serviceModel, GlobalBusProvider␣
↪globalBusProvider) {
    super(translate, serviceModel, globalBusProvider);
}
```

(continues on next page)

(continued from previous page)

```
@Override
protected void doStop() throws Exception {

}

@Override
protected void doStart() throws Exception {

}

@Override
public I18NKey getNameKey() {
    return LabelKey.ServiceA;
}
```

}

```
[source]
----
package com.example.tutorial.service;

import com.example.tutorial.i18n.LabelKey;
import com.google.inject.Inject;
import uk.q3c.krail.eventbus.GlobalBusProvider;
import uk.q3c.krail.service.AbstractService;
import uk.q3c.krail.service.ServiceModel;
import uk.q3c.krail.i18n.I18NKey;
import uk.q3c.krail.i18n.Translate;

@Singleton
public class ServiceB extends AbstractService {

    @Inject
    protected ServiceB(Translate translate, ServicesModel serviceModel,␣
↪GlobalBusProvider globalBusProvider) {
        super(translate, serviceModel, globalBusProvider);
    }

    @Override
    protected void doStop() throws Exception {

    }

    @Override
    protected void doStart() throws Exception {

    }

    @Override
    public I18NKey getNameKey() {
        return LabelKey.ServiceB;
    }
}

----

[source]
```

(continues on next page)

(continued from previous page)

```
----
package com.example.tutorial.service;

import com.example.tutorial.i18n.LabelKey;
import com.google.inject.Inject;
import uk.q3c.krail.eventbus.GlobalBusProvider;
import uk.q3c.krail.service.AbstractService;
import uk.q3c.krail.service.ServiceModel;
import uk.q3c.krail.i18n.I18NKey;
import uk.q3c.krail.i18n.Translate;

@Singleton
public class ServiceC extends AbstractService {

    @Inject
    protected ServiceC(Translate translate, ServicesModel serviceModel,␣
↪GlobalBusProvider globalBusProvider) {
        super(translate, serviceModel, globalBusProvider);
    }

    @Override
    protected void doStop() throws Exception {

    }

    @Override
    protected void doStart() throws Exception {

    }

    @Override
    public I18NKey getNameKey() {
        return LabelKey.ServiceC;
    }
}

----

[source]
----
package com.example.tutorial.service;

import com.example.tutorial.i18n.LabelKey;
import com.google.inject.Inject;
import uk.q3c.krail.eventbus.GlobalBusProvider;
import uk.q3c.krail.service.AbstractService;
import uk.q3c.krail.service.ServiceModel;
import uk.q3c.krail.i18n.I18NKey;
import uk.q3c.krail.i18n.Translate;

@Singleton
public class ServiceD extends AbstractService {

    @Inject
    protected ServiceD(Translate translate, ServicesModel serviceModel,␣
↪GlobalBusProvider globalBusProvider) {
        super(translate, serviceModel, globalBusProvider);
```

(continues on next page)

(continued from previous page)

```
    }

    @Override
    protected void doStop() throws Exception {

    }

    @Override
    protected void doStart() throws Exception {

    }

    @Override
    public I18NKey getNameKey() {
        return LabelKey.ServiceD;
    }
}

----

Note that each has a different name key - this is also used by getServiceKey(), which␣
↪is used to uniquely identify a Service class. This approach is used to overcome the␣
↪changes in class name which occur when using enhancers such as Guice AOP. This␣
↪means that each Service class must have a unique name key.

As Services often are, these are all Singletons, although they do not have to be.

== Registering Services

All Service classes must be registered. We can do that very simply by sub-classing␣
↪`AbstractServiceModule` and using the methods it provides

* create a new class `TutorialServicesModule` in _com.example.tutorial.service_
* copy the code below
[source]
----
package com.example.tutorial.service;

import com.example.tutorial.i18n.LabelKey;
import uk.q3c.krail.service.AbstractServiceModule;
import uk.q3c.krail.service.Dependency;

public class TutorialServicesModule extends AbstractServiceModule {

    @Override
    protected void registerServices() {
        registerService(LabelKey.ServiceA, ServiceA.class);
        registerService(LabelKey.ServiceB, ServiceB.class);
        registerService(LabelKey.ServiceC, ServiceC.class);
        registerService(LabelKey.ServiceD, ServiceD.class);
    }

    @Override
    protected void defineDependencies() {

    }
}
```

(continues on next page)

(continued from previous page)

```
----

* include the module in the `BindingManager`:
[source]
----
@Override
protected void addAppModules(List<Module> baseModules) {
    baseModules.add(new TutorialServicesModule());
}

----

== Monitor the Service status

Fur the purposes of the Tutorial, we will create a simple page to monitor the status␣
↪of the Services.

* In `MyOtherPages` add the entry:
[source,java]
----
addEntry("services", ServicesView.class, LabelKey.Services, PageAccessControl.PUBLIC);
----

* create `ServicesView` in the _com.example.tutorial.pages_ package
[source]
----
package com.example.tutorial.pages;

import com.example.tutorial.i18n.Caption;
import com.example.tutorial.i18n.DescriptionKey;
import com.example.tutorial.i18n.LabelKey;
import com.example.tutorial.service.ServiceA;
import com.example.tutorial.service.ServiceB;
import com.example.tutorial.service.ServiceC;
import com.example.tutorial.service.ServiceD;
import com.google.inject.Inject;
import com.vaadin.ui.Button;
import com.vaadin.ui.Panel;
import com.vaadin.ui.TextArea;
import com.vaadin.ui.VerticalLayout;
import net.engio.mbassy.listener.Handler;
import net.engio.mbassy.listener.Listener;
import uk.q3c.krail.eventbus.GlobalBus;
import uk.q3c.krail.eventbus.SubscribeTo;
import uk.q3c.krail.service.ServiceBusMessage;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;
import uk.q3c.krail.i18n.Translate;

@Listener
@SubscribeTo(GlobalBus.class)
public class ServicesView extends Grid3x3ViewBase {

    private ServiceA serviceA;
    private ServiceB serviceB;
    private final ServiceC serviceC;
    private final ServiceD serviceD;
```

(continues on next page)

(continued from previous page)

```
    @Caption(caption = LabelKey.Start_Service_A, description = DescriptionKey.Start_
↪Service_A)
    private Button startABtn;
    @Caption(caption = LabelKey.Stop_Service_D, description = DescriptionKey.Stop_
↪Service_D)
    private Button stopDBtn;
    @Caption(caption = LabelKey.Stop_Service_C, description = DescriptionKey.Stop_
↪Service_C)
    private Button stopCBtn;
    @Caption(caption = LabelKey.Stop_Service_B, description = DescriptionKey.Stop_
↪Service_B)
    private Button stopBBtn;
    private Translate translate;
    @Caption(caption = LabelKey.State_Changes,description = DescriptionKey.State_
↪Changes)
    private TextArea stateChangeLog;
    @Caption(caption = LabelKey.Clear,description = DescriptionKey.Clear)
    private Button clearBtn;

    @Inject
    protected ServicesView(Translate translate,ServiceA serviceA, ServiceB serviceB,␣
↪ServiceC serviceC, ServiceD serviceD) {
        super(translate);
        this.translate = translate;
        this.serviceA = serviceA;
        this.serviceB = serviceB;
        this.serviceC = serviceC;
        this.serviceD = serviceD;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        createButtons();
        createStateMonitor();

    }

    private void createStateMonitor() {
        stateChangeLog = new TextArea();
        stateChangeLog.setSizeFull();
        stateChangeLog.setRows(8);
        getGridLayout().addComponent(stateChangeLog,0,1,2,1);
        clearBtn = new Button();
        clearBtn.addClickListener(click->stateChangeLog.clear());
        setBottomCentre(clearBtn);
    }

    @Handler
    protected void handleStateChange(ServiceBusMessage serviceBusMessage) {
        String serviceName = translate.from(serviceBusMessage.getService()
                                                              .getNameKey());
        String logEntry = serviceName + " changed from " + serviceBusMessage.
↪getFromState()
                                                                      .name() +
↪" to " + serviceBusMessage.getToState().name()+", cause: " +
```

(continues on next page)

(continued from previous page)

```
                serviceBusMessage.getCause();
        String newline = stateChangeLog.getValue().isEmpty() ? "" : "\n";
        stateChangeLog.setValue(stateChangeLog.getValue()+newline+logEntry);
    }

    private void createButtons() {
        startABtn = new Button();
        startABtn.addClickListener(click -> serviceA.start());

        stopDBtn = new Button();
        stopDBtn.addClickListener(click -> serviceD.stop());

        stopCBtn = new Button();
        stopCBtn.addClickListener(click -> serviceC.stop());

        stopBBtn = new Button();
        stopBBtn.addClickListener(click -> serviceB.stop());

        Panel panel = new Panel();
        VerticalLayout layout = new VerticalLayout(startABtn, stopDBtn, stopCBtn,␣
↪stopBBtn);
        panel.setContent(layout);
        setTopLeft(panel);
    }
}
----

* create the enum constants

Here we set up some buttons to start and stop services in `createButtons()`<br>
We use the link:tutorial-event-bus.md[Event Bus] to create a simple monitor for state␣
↪changes in `createStateMonitor()`

* run the application and try pressing 'Start Service A' - a message will appear in␣
↪the state changes log

== Defining Dependencies

So far, all the Services operate independently - there are no dependencies specified.␣
↪Let us assume we want service A to depend on the other 3 services, each with a␣
↪different one of the 3 dependency types. We will also mix up using Guice and␣
↪*Dependency* annotations, though you would probably use only one method to avoid␣
↪confusion.

=== Dependencies with Guice

* add the following to the `defineDependencies()` method in the␣
↪`TutorialServicesModule`:
[source,java]
----
addDependency(LabelKey.ServiceA,LabelKey.ServiceB, Dependency.Type.ALWAYS_REQUIRED);
addDependency(LabelKey.ServiceA,LabelKey.ServiceC, Dependency.Type.REQUIRED_ONLY_AT_
↪START);
----

=== Dependencies by Annotation
```

(continues on next page)

(continued from previous page)

```
In `ServiceA` we inject `ServiceD` and store in a field in order to annotate it as a␣
↪dependency (which you would need anyway if you wish to access `ServiceD`).

* Modify ServiceA
[source,java]
----

    @Dependency(required = false)
    private ServiceD serviceD;

    @Inject
    protected ServiceA(Translate translate, ServicesModel serviceModel,␣
↪GlobalBusProvider globalBusProvider, ServiceD serviceD) {
        super(translate, serviceModel, globalBusProvider);
        this.serviceD = serviceD;
    }

----

This marks the dependency, ServiceD, as optional

== Testing Dependencies

* run the application
* navigate to the 'Services' page
* press 'Start Service A'
* Note that all 4 services show in the state changes log as 'STARTED' - `ServiceA`␣
↪has automatically called all its dependencies to start. The order they start in is␣
↪arbitrary, as they are started in parallel threads, but `ServiceA` will not start␣
↪until all its required dependencies have started.
* press 'Clear'
* press 'Start Service A' again - nothing happens. Attempts to start/stop a service␣
↪which is already started/stopped are ignored.
* press 'Stop ServiceD' - only `ServiceD` stops
* press 'Stop ServiceC' - only `ServiceC` stops
* press 'Stop ServiceB' - `ServiceB` and `ServiceA` stop. `ServiceA` has cause of␣
↪DEPENDENCY_STOPPED

When `ServiceD` and `ServiceC` are stopped they do not affect `ServiceA`, as they are␣
↪declared as "optional" and "required only at start".
When `ServiceB` is stopped, however, `ServiceA` also stops because that dependency␣
↪was declared as "always required"

= Summary

* We have created services by sub-classing `AbstractService`
* We have defined dependencies between services using Guice
* We have defined dependencies between services using the *@Dependency* annotation
* We have demonstrated the interaction between services, when starting and stopping␣
↪services with different dependency types

= Download from GitHub

To get to this point straight from GitHub, https://github.com/davidsowerby/krail-
↪tutorial[clone] using branch *step11*
```

CHAPTER 15

Push

You may recall from the Event Bus chapter that a Vaadin client is unaware of changes made on the server. We had to force the client to poll the server for updates by clicking a button.

To overcome this, Vaadin introduced 'Push' in version 7.1, a feature used to push messages from server to client.

Krail implements the process described in the Vaadin Handbook and extends it slightly:

- a Broadcaster is implemented to enable any registered UI to push messages
- ScopedUI automatically registers with the Broadcaster, so that any UI can push a message
- ScopedUI listens for broadcast messages and distributes them via the UI Event Bus as instances of `PushMessage`

15.1 Fixing the Refresh Problem

15.1.1 Modify the UI

- Add a **@Push** annotation to the `TutorialUI`

```
@Theme("valo")
@Push
public class TutorialUI extends DefaultApplicationUI {
```

15.1.2 Broadcast a message

- remove the refresh button (we will no longer need that), and its **@Caption**

When we press the Send Message buttons, we want to push a message as well. In `EventsView`:

- inject the `Broadcaster` and make it a field:
- modify each button click listener to broadcast (push) a message with a call to `broadcaster.broadcast()`:

```
package com.example.tutorial.pages;

import com.example.tutorial.eventbus.*;
import com.example.tutorial.i18n.Caption;
import com.example.tutorial.i18n.DescriptionKey;
import com.example.tutorial.i18n.LabelKey;
import com.google.inject.Inject;
import com.vaadin.ui.Button;
import uk.q3c.krail.eventbus.GlobalBusProvider;
import uk.q3c.krail.core.eventbus.SessionBusProvider;
import uk.q3c.krail.core.eventbus.UIBusProvider;
import uk.q3c.krail.i18n.Translate;
import uk.q3c.krail.core.push.Broadcaster;
import uk.q3c.krail.core.view.Grid3x3ViewBase;
import uk.q3c.krail.core.view.component.ViewChangeBusMessage;

public class EventsView extends Grid3x3ViewBase {
    private final UIBusProvider uiBusProvider;
    private final GlobalBusProvider globalBusProvider;
    private Broadcaster broadcaster;
    @Caption(caption = LabelKey.Singleton, description = DescriptionKey.Singleton)
    private Button singletonSendBtn;
    @Caption(caption = LabelKey.Session, description = DescriptionKey.Session)
    private Button sessionSendBtn;
    @Caption(caption = LabelKey.UI, description = DescriptionKey.UI)
    private Button uiSendBtn;
    private SessionBusProvider sessionBusProvider;
    private GlobalMessageReceiver singletonMessageReceiver;
    private MessageReceiver sessionMessageReceiver;
    private MessageReceiver uiMessageReceiver;

    @Inject
    protected EventsView(Translate translate, UIBusProvider uiBusProvider,␣
↪SessionBusProvider sessionBusProvider, GlobalBusProvider globalBusProvider,
                         GlobalMessageReceiver singletonMessageReceiver,␣
↪SessionMessageReceiver sessionMessageReceiver, UIMessageReceiver uiMessageReceiver,
                         Broadcaster broadcaster) {
        super(translate);
        this.uiBusProvider = uiBusProvider;
        this.sessionBusProvider = sessionBusProvider;
        this.singletonMessageReceiver = singletonMessageReceiver;
        this.sessionMessageReceiver = sessionMessageReceiver;
        this.uiMessageReceiver = uiMessageReceiver;
        this.globalBusProvider = globalBusProvider;
        this.broadcaster = broadcaster;
    }

    @Override
    protected void doBuild(ViewChangeBusMessage busMessage) {
        super.doBuild(busMessage);
        constructEventSendButtons();
        layoutReceivers();
    }

    private void layoutReceivers() {
        setTopCentre(singletonMessageReceiver);
        setMiddleCentre(sessionMessageReceiver);
```

(continues on next page)

(continued from previous page)

```
        setBottomCentre(uiMessageReceiver);
    }

    private void constructEventSendButtons() {
        singletonSendBtn = new Button();
        sessionSendBtn = new Button();
        uiSendBtn = new Button();
        singletonSendBtn.addClickListener(click -> {
            String m = "Singleton";
            globalBusProvider.get()
                             .publish(new TutorialMessage(m, this));
            broadcaster.broadcast("refresh", m, this.getRootComponent());

        });
        sessionSendBtn.addClickListener(click -> {
            String m = "Session";
            sessionBusProvider.get()
                              .publish(new TutorialMessage(m, this));
            broadcaster.broadcast("refresh", m, getRootComponent());
        });
        uiSendBtn.addClickListener(click -> {
            String m = "UI";
            uiBusProvider.get()
                         .publish(new TutorialMessage(m, this));
            broadcaster.broadcast("refresh", m, getRootComponent());
        });
        setTopLeft(singletonSendBtn);
        setMiddleLeft(sessionSendBtn);
        setBottomLeft(uiSendBtn);
    }
}
```

15.1.3 Verifying the change

We will now do the same sequence of tasks as for the Event Bus, but without pressing the refresh button

- refresh Gradle
- run the application
- open a browser, which we will call browser 1 tab 1
- login as *'admin'*, *'password'*
- navigate to the *Event Bus* page
- open a second browser tab at the same URL - we will call this browser 1 tab 2
- in browser 1 tab 1 press each of the 3 buttons, Singleton, Session and UI
- Messages will appear in all 3 text areas
- Switch to tab 2
- the Singleton and Session text areas will contain a message from the same source, but the UI area will be empty

This demonstrates the scope of the event buses. The UI bus is of UIScope - which means it relates to a browser tab (unless embedded). The session scope relates to a browser instance, and therefore appears in both tabs, and a singleton scope applies to an application and also appears in both tabs.

- open a second browser instance (if you are using Chrome, be aware that Chrome does odd things with browser instances - to be certain you have a separate instance, it is better to use Firefox as the second instance)
- in browser 2, login as *'admin'*, *'password'*
- navigate to the *Event Bus* page
- switch back to browser 1 tab 1 and press each of the 3 buttons, Singleton, Session and UI again
- switch browser 2 tab 1
- Only the Singleton text area will contain a message, as expected

15.2 Using a Push Message

You may have noticed that we did not actually use the `PushMessage`, just broadcasting it was enough to prompt the client to poll changes from the server. We could, however, pick them up and use them as they are captured by the `ScopedUI` and despatched via the UI Bus. To demonstrate this we will simply show the push messages in the UI state change log:

- Modify `MessageReceiver` by adding a getter

```
public TextArea getTextField() {
    return textField;
}
```

- Modify `UIMessageReceiver` to capture `PushMessage` instances and update the state change log:

```
package com.example.tutorial.eventbus;

import net.engio.mbassy.listener.Handler;
import net.engio.mbassy.listener.Listener;
import uk.q3c.krail.eventbus.SubscribeTo;
import uk.q3c.krail.core.eventbus.UIBus;
import uk.q3c.krail.core.push.PushMessage;

@Listener
@SubscribeTo(UIBus.class)
public class UIMessageReceiver extends MessageReceiver {

    @Handler
    public void pushMessage(PushMessage pushMessage) {
        String s = getText();
        getTextField().setValue(s + "\n" + "Push message was originally from:
↪"+pushMessage.getMessage());
    }
}
```

- run the application
- press any of the send message buttons, and an additional "push" message will appear in all the UI state log texts, of any UIs (browser tabs) you have open

15.3 Footnote

Vaadin Push can be a little quirky. This Tutorial was developed using Tomcat 8, and also checked on Tomcat 7 - but if you use something else and get problems, it is worth checking Vaadin's notes on the subject first.

15.4 Summary

- We have broadcast a push message and seen that it causes the client to poll for updates, enabling immediate client refresh from a server based change.
- we have intercepted the push message after it has been re-distributed via the UI Bus

15.5 Download from GitHub

To get to this point straight from GitHub, clone using branch **step12**

CHAPTER 16

Create a project Using Eclipse

16.1 Acknowledgement

Thanks to Dirk Lietz for contributing this chapter. If you have any questions regarding this chapter please refer them to the contributor

16.2 Introduction

A short how-to set up krail as a library in a new Vaadin-Project in Eclipse

16.3 Install Vaadin-Plugin

Install the Vaadin Plugin for Eclipse

16.4 Create a new Vaadin Project

File -> New -> Other . . . Vaadin -> Vaadin 7 Project

Give it a Name and select the Target-Runtime (Apache Tomcat v8) and Java 1.8 Select the Deployment configuration : Servlet (default)

Hit Button Finish (or Next to configure some more Details like Pakage Names)

A new Vaadin-Project will now be created with ivy-dependency Management set up

16.5 Apply Krail-Dependency

Open `ivysettings.xml` and add the jcenter repository in the `<resolvers>` section: ‘xml <!– jcenter –> <ibiblio name=”jcenter” root=”http://jcenter.bintray.com" m2compatible=”true”/> ‘

Open `ivy.xml` and add the krail-library in the `<dependencies>` section ‘xml <!– The core of krail –> <dependency org=”uk.q3c.krail” name=”krail” rev=”0.9.3” /> ‘

The whole `ivy.xml` file could look like (with krail-kpa add on set up):

```
&lt;?xml version="1.0"?&gt;
&lt;!DOCTYPE ivy-module [
 &lt;!ENTITY vaadin.version "7.4.6"&gt;
 &lt;!ENTITY krail.version "0.9.3"&gt;
 &lt;!ENTITY krail-jpa.version "0.8.8"&gt;
]&gt;
&lt;ivy-module version="2.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance[http://www.w3.org/2001/
↪XMLSchema-instance]"
 xsi:noNamespaceSchemaLocation=&amp;quot;http://ant.apache.org/ivy/schemas/ivy.xsd"[http:/
↪/ant.apache.org/ivy/schemas/ivy.xsd"]
 <info organisation="com.example" module="tutorial" />
 <configurations>
 <!-- The default configuration, which should be deployed to the server -->
 <conf name="default" />
 <!-- A configuration only needed when compiling the widget set. Should
            not be deployed to the server -->
 <conf name="widgetset-compile" />
 <!-- A configuration used in compilation of server side classes only.
            Should be deployed to the server -->
 <conf name="nodeploy" />
 </configurations>
 <dependencies defaultconf="default" defaultconfmapping="default->default">
 <!-- The core of krail -->
 <dependency org="uk.q3c.krail" name="krail" rev="&amp;krail.version;" />

[source]
----
    <!-- Add-On krail-jpa -->
    <dependency org="uk.q3c.krail" name="krail-jpa" rev="&amp;krail-jpa.version;" />

    <!-- The core server part of Vaadin -->
    <dependency org="com.vaadin" name="vaadin-server" rev="&amp;vaadin.version;" />

    <!-- Vaadin themes -->
    <dependency org="com.vaadin" name="vaadin-themes" rev="&amp;vaadin.version;" />

    <!-- Push support -->
    <dependency org="com.vaadin" name="vaadin-push" rev="&amp;vaadin.version;" />

    <!-- Servlet 3.0 API -->
    <dependency org="javax.servlet" name="javax.servlet-api" rev="3.0.1" conf=
↪"nodeploy->default" />

    <!-- Precompiled DefaultWidgetSet -->
    <dependency org="com.vaadin" name="vaadin-client-compiled"
       rev="&amp;vaadin.version;" />
```

(continues on next page)

(continued from previous page)

```
    <!-- Vaadin client side, needed for widget set compilation -->
    <dependency org="com.vaadin" name="vaadin-client" rev="&vaadin.version;"
        conf="widgetset-compile->default" />

    <!-- Compiler for custom widget sets. Should not be deployed -->
    <dependency org="com.vaadin" name="vaadin-client-compiler"
       rev="&vaadin.version;" conf="widgetset-compile->default" />
</dependencies>
----

</ivy-module>
```

CHAPTER 17

Create a Hierarchy

to be written

CHAPTER 18

Functional Testing

Krail provides some support for Functional Testing out of the box. This is still in development, but there may be parts of it you may find useful.

See *Functional Testing* in the Developer Guide for more detail.

CHAPTER 19

Introduction to the User Guide

This document aims to assist the Krail developer to understand how to develop an application based on Krail, by dipping in and out of features as required.

A *Tutorial* is available to walk through the process, step by step.

If you are are interested in the reasons for some of the development decisions made in Krail, take a look at the *Developer Guide.*.

CHAPTER 20

Bootstrap

In order to allow the same application code to run in both Vertx and Servlet environments, a bootstrap sequence is used. This is only really needed so that the Guice Injector can be held in a location suitable for the environment, so that it can be retrieved during *deserialisation.* As a side effect, it has the additional benefit of simplifying basic application configuration.

20.1 Bootstrap File

A file called 'krail-bootstrap.yml' should be placed in *src/main/resources*.

20.1.1 Sample File

A minimal example file.

```
version: 1
collator: uk.q3c.krail.core.guice.CoreBindingsCollator
modules:
  - com.example.myapp.MyAppModule1
servlet:
 modules:
  - uk.q3c.krail.core.env.ServletEnvironmentModule
vertx:
  modules:
   - uk.q3c.krail.core.env.VertxEnvironmentModule
```

20.1.2 File Content

version

Optional, defaults to 1.

collator

Required. A fully qualified reference to your implementation of `BindingsCollator`, or you could even leave it as *uk.q3c.krail.core.guice.CoreBindingsCollator* and add your own modules as below.

modules

Optional. Fully qualified references to modules you want to add to the collator. These will apply to both Servlet and Vertx environments.

servlet

Required if you intend to run the application in a Servlet environment.

servlet/modules

Required if you intend to run the application in a Servlet environment. Modules to be added to the collator only for the Servlet environment. Unless its bindings are replaced elsewhere you will need at least *uk.q3c.krail.core.guice.ServletEnvironmentModule* as shown in the example.

vertx

Required if you intend to run the application in a Vertx environment.

vertx/modules

Required if you intend to run the application in a Vertx environment. Modules to be added to the collator only for the Vertx environment. Unless its bindings are replaced elsewhere you will need at least *uk.q3c.krail.core.guice.VertxEnvironmentModule* as shown in the example.

CHAPTER 21

Injector Scope

The scope of the Guice Injector can be significant. A **@Singleton** is described by Guice as "per application", and is therefore per Injector.

In most situations the injector scope is per JVM ClassLoader - the same as a static variable.

Of course, different environments treat ClassLoaders in different ways.

The important part for Krail is the "per application" definition for Guice.

21.1 Accessing the Injector

It is normally considered bad practice to access the injector directly - the whole point of IoC is to hand over control. But as always there are special cases.

In Krail there are two scenarios where it is considered reasonable to access the injector directly:

21.1.1 Deserialisation

`SerializationSupport` is used to re-inject Guice supplied *transient dependencies*, following deserialisation. Deserialisation occurs without any reference to Guice of course, so without this intervention, transient dependencies would be **null**.

21.1.2 View and UI Factory

The `MasterSitemap` is a central component of Krail, and it uses `KrailView` classes as part of the site definition. At the moment, the most practical way to deal with this is to instantiate these views with the injector, when they are needed.

When support for multiple views and UIs per route is implemented, dynamic construction based on potentially any selection criteria will be required. There may be a better way, but currently it is looking like this will continue to need access to the injector as well. (See issues 664 and 665)

CHAPTER 22

Serialization

When Vaadin serialises to the session, it serialises the entire UI. This means anything contained within the UI is also serialized. If you follow the Krail approach of constructor injection for Views and UIs, it will mean that those dependencies will also be serialized, unless, of course, they are marked as **transient**.

This clearly could affect the amount that needs to be serialized/deserialized - you may want to reduce that by making dependencies **transient** (or you may just have dependencies which cannot be serialized), but that in turn means you need a way to reconstruct the **transient** fields.

22.1 Serialization and Shiro / JPA

Anything which uses Guice AOP generates a byte enhanced class produced by cglib. This causes serialization problems, and is a feature of anything which uses cglib. At the moment the only way round this is to use manual coding instead of AOP supported annotations. For example, instead of using:

```
@RequiresPermission()
public void doSomething(){

}
```

use

```
if (subjectProvider.get().isPermitted("page:view:private")) {
    userNotifier.notifyInformation(LabelKey.Yes);
}
```

There is an open issue to provide more support.

22.2 Guice Deserialization for View and UI instances

`ViewBase` and `ScopedUI` use `SerializationSupport` to make the management of this situation simpler, designed in a way for sub-classes to make use of this facility.

When instances or sub-classes of `ViewBase` and `ScopedUI` are deserialized, a standard Java `readObject()` is invoked method, and `SerializationSupport` used to re-inject **transient** fields using the Guice Injector. Hooks are also provided to allow you to intervene with your own logic at various points.

Caution: To enable this to work, certain conditions apply. Sub-classes of `ViewBase` and `ScopedUI` :

- must have non-Serializable fields must be marked **transient**, as normal
- will only attempt to re-inject transient fields which have a null value at the time it invokes `SerializationSupport.injectTransientFields()` - see the call sequence below
- must have an exact match between the type of the constructor parameter and the type of the field that it is associated with
- will raise an exception if, after completing the sequence of calls below, there are still null **transient** fields

22.2.1 Call Sequence

This is the sequence of calls made during deserialization. Note that *injection by SerializationSupport* will only inject into null **transient** fields

- beforeDeserialization()
- *default deserialization*
- beforeTransientInjection()
- *SerializationSupport injects transients*
- afterTransientInjection()
- *SerializationSupport checks for null transients*, and raises exception if any found (unless excluded)

22.2.2 Matching constructor parameters with fields

In order to match a constructor parameter with its field for automatic re-injection, they must both be of exactly the same type (and not just assignable). In the case of Guice, the type includes the use of binding annotations.

This means that where a binding annotation is used on a constructor parameter, its associated field must also have the same binding annotation.

Java example

In Java, we must annotate the field to match a constructor parameter that uses a binding annotation. Your IDE may flag a warning that you have a binding annotation without @Inject - this can be ignored / suppressed. If you do annotate the field with @Inject, then outside of deserialization, Guice will inject the field twice, once via the constructor, and once directly to the field.

```
public class MyView extends ViewBase {

   @Named("1)  // to match its constructor parameter
   private transient Widget widget1;

   @Inject
   protected MyView(Translate translate, SerializationSupport serializationSupport,␣
↪@Named("1") Widget widget1){
```

(continues on next page)

(continued from previous page)

```
        super(translate, serializationSupport)
        this.widget1=widget1
    }

}
```

Kotlin example

Because Kotlin declares a property rather than a separate constructor parameter and field, the property needs to be annotated in a way that causes Kotlin's code generator to correctly annotate its Java output:

```
class MyView @Inject constructor(translate:Translate,␣
↪serializationSupport:SerializationSupport, @field:Named("1") @param:Named("1")␣
↪@Transient val widget1:Widget) : ViewBase(translate,serializationSupport)
```

22.2.3 Excluding fields

If for some reason you want a transient field to be null at the end of the deserialization process, fields can be excluded from injection and the final check, by overriding the `ViewBase` or `ScopedUI` method `beforeDeserialization()` or `beforeTransientInjection()` to set the exclusions

```
protected void beforeTransientInjection(){
    serializationSupport.setExcludedFieldNames(ImmutableList.of("thisField"));
}
```

Tip: Guice, Binding Annotations and Inheritance. There is an "interesting" side effect from using Guice binding annotations. It is very easy to provide the binding on a superclass constructor parameter, and then forget to put it on the equivalent sub-class constructor parameter - meaning you have injected something different via the sub-class. Your IDE and compiler will not tell you. This Serialization routine will tell you if you do so. This was not really a design choice, just a bit of luck!

22.3 Non-Serializable classes

This list is not exhaustive, but identifies some of the commonly used Krail classes which cannot be made Serializable. For these, use the method described above to re-inject them.

- `BusProvider` implementations which use MBassador. This currently applies to all `BusProvider` implementations.
- `PubSubSupport` from MBassador

22.4 Making your classes 'Guice Serializable'

Where you need to deserialize your own classes that are constrcuted by Guice, but has non-Serializable dependencies, you can still use `SerializationSupport`, within the standard `readObject()` deserialization method:

Listing 1: Java

```
private void readObject(ObjectInputStream inputStream) throws ClassNotFoundException,
↪ IOException {
    inputStream.defaultReadObject();
    serializationSupport.deserialize(this);
}
```

Listing 2: Kotlin

```
@Throws(ClassNotFoundException::class, IOException::class)
private fun readObject(inputStream: ObjectInputStream) {
    inputStream.defaultReadObject()
    serializationSupport.deserialize(this)
}
```

This combines the calls above, and invokes `defaultReadObject()`, `injectTransients()` and `checkForNullTransients()` If you want to exclude any fields, just set `serializationSupport.excludedFieldNames` before invoking `deserialize()`

Usually this happens when an object is created by a factory which then supplies Guice-constructed dependencies and some stateful element to the constructor - this is typical of a situation which Guice `AssistedInject` is used.

Caution: Some tests failed when using Guice `AssistedInject` with Serialization - we avoid using it, and use manually coded factories instead where needed. To be fair though, we are not completely sure there is a real problem, see open issue

Listing 3: Java

```
public class MyObjectFactory{

    public MyObjectFactory (String statefulElement, MyNonSerializableDependency␣
↪dependency){
     //etc
    }

    private void readObject(ObjectInputStream inputStream) throws␣
↪ClassNotFoundException, IOException {
        inputStream.defaultReadObject();
        this.dependency = serializationSupport.getInjector().
↪getInstance(MyNonSerializableDependency.class);
    }

}
```

CHAPTER 23

Forms

23.1 Overview

Vaadin provides some support for Forms with `Binder`, but Krail takes that further. It makes the definition of a Form part of the Sitemap by assigning a `FormConfiguration` to a `SitemapNode`.

The Form class takes that configuration and builds the form with UI components (TextField etc) and integrates Krail's I18N and JSR 303 validation.

Two Form types are currently provided:

- **simple**, which displays/edits selected properties form a given entity class
- **list**, which displays a *table* of selected properties, for one more instance of the same entity class

Additional form types can easily be added.

23.2 Defining a Form

To construct a form in Krail (using a Person entity as an example):

1. Define your form configuration as a sub-class of `FormConfiguration`, for example `PersonFormConfiguration`
2. using either the Direct or Annotation method of creating a `Sitemap` entry, set the *viewClass* to `Form.class`
3. set the *viewConfiguration* to `PersonFormConfiguration.class`

23.3 Form construction

As part of Krail's navigation process, the view for a given URI is looked up from the Sitemap. The *viewClass* is constructed via Guice, and an instance of the *viewConfiguration* passed to it (in this example an instance of `PersonFormConfiguration`

1. `Form` invokes `FormTypeSelector` to acquire the correct `FormBuilder`
2. `FormBuilder` uses `FormConfiguration` in combination with `FormSupport` to construct appropriate UI components (TextFields etc) and bind them to enity data. The binding is carried out by `KrailBeanValidationBinder`, which also takes care of integrating JSR303 validation and I18N.

23.3.1 Validation

Validation can be defined by JSR303 annotations on the entity or directly within the `FormConfiguration`

23.4 Model to Presentation mapping

`FormSupport` provides the mappings of data types to presentation Fields, along with data converters. These mappings are defined in Guice and can therefore be easily extended or overruled.

23.4.1 Defaults

For each property (the Model) that is being bound to the user interface, a component (the Presentation) is needed. To enable the automatic creation of presentation elements, `FormSupport.fieldFor()` uses a map of data types to Vaadin Fields - for example, a String is mapped to a `TextField`.

Default mappings are provided by the `FormModule`, but these can be overridden for specific instances within `FormConfiguration`.

23.4.2 Changing defaults

To change the default Model to Presentation mappings, sub-class `FormModule` override `bindDefaultDataClassMappings`, and replace `FormModule` with your sub-class in your `BindingsCollator`

23.4.3 Register a new mapping

A new data type can be registered, by creating another Guice module which contributes a `MapBinder` as below

Listing 1: Kotlin

```
class MyMappingModule : AbstractModule() {

  override fun configure() {
     val fieldLiteral = object : TypeLiteral<AbstractField<*>>() {}
     val dataClassLiteral = object : TypeLiteral<Class<*>>() {}
     val dataClassToFieldMap: MapBinder<Class<*>, AbstractField<*>> = MapBinder.
↪newMapBinder(binder(), dataClassLiteral, fieldLiteral)

     // bind new data types
     dataClassToFieldMap.addBinding(MyDataClass::class.java).to(WidgetField::class.
↪java)
 }
```

23.5 Model to Presentation Converters

Where the model and presentation type are the same, clearly no conversion is needed, although a `NoConversionConverter` is actually used to transfer the data.

Where a converter is needed - for example, to display an integer in a TextField, a `StringToIntegerConverter` is needed - this converter type is provided by `FormSupport.converterFor()`

Ultimately this uses an implementation of `ConverterFactory` to instantiate the converter itself. The default implementation `DefaultConverterFactory`, iterates through all `ConverterSets` classes defined via Guice until it finds one to match the desired model and presentation class, or throws an exception if none found.

By default there is just one `ConverterSet`, the `BaseConverterSet`.

23.5.1 Adding / Replacing Converters

Converters can be added by defining your own `ConverterSet`, and adding it in one of two ways:

- sub-class `ConverterModule` and override the `define()` method to provide bindings to additional `ConverterSet` implementations, and replace `ConverterModule` in your `BindingsCollator`
- create a new module using `ConverterModule` as an example (in particular the MultiBinder), and add the new module to your `BindingsCollator`

CHAPTER 24

License

Krail is licensed under Apache 2.0

CHAPTER 25

Introduction to the Developer Guide

This guide is aimed at those who are developing Krail itself, or are just interested in the reasons behind some of the design choices made.

25.1 Accuracy

Please note that some of this this guide is still a fairly random collection of notes, and has not kept up to date with releases.

25.1.1 Up to date sections

These sections, however, are considered to be up to date:

- *Goals*
- *Bootstrap*
- *Push*
- *Serialisation*
- *Vertx*

CHAPTER 26

Goals and Objectives

26.1 Terminology

- **Krail developer** - someone developing an application based on Krail
- **Krail team** - the team which writes Krail itself

26.2 Goals

1. To produce a framework which enables a Krail developer to produce a reliable and complete business application quickly which:
   (a) is easy to change and adapt to changing requirements
   (b) enables the Krail developer to concentrate on business requirements rather than technical requirements
   (c) re-uses existing, well proven code wherever possible
   (d) supports a microservices architecture
   (e) supports a traditional servlet based architecture

26.3 Objectives

1. provide a microservices architecture using Eclipse Vert.x
2. enable the use of the same code in a servlet environment with minimal configuration changes
3. allow the Krail developer to develop in just Java or Kotlin as much as possible, minimising the need for CSS, HTML, XML etc

26.4 Priorities

1. A Vert.x solution is a higher priority than the Servlet solution. If compromises are absolutely necessary, then it is the Servlet solution which should be adjusted.

26.5 Krail Team Goals

These goals relate only to Krail itself, not the application developed on Krail. They are still goals, but considered less important than the "business" goals described above.

1. Everything in a single language - the current mix of Java, Kotlin and Groovy should migrate eventually to Kotlin only. This applies to source, test code and build scripts.
2. A common test framework. There is currently a mix of Junit Java, JUnit Kotlin and Spock. Ideally this will all migrate to Spek (Kotlin) - but may have to include JUnit Kotlin to enable Vertx testing.
3. Kotlin based build. This could be either Gradle with Kotlin script (in place of Groovy scripts) or Kobalt. Kobalt needs to be assessed before making a switch
4. A single development lifecycle, but with optional steps. This is currently provided by the KayTee plugin, but that is not at production standard. It is, however, a companion product to Krail

26.5.1 Priorities

1. Migrating to an all Kotlin solution is not urgent. It can be carried out when the opportunity arises.
2. New code can be accepted in Java if it offers new or improved functionality.
3. New tests should ideally be in Spek but note the limitation below. JUnit tests in Kotlin or Java can also be accepted if really necessary.

26.5.2 Spek Limitation

The current version of Spek (1.1.5) does not play well with JUnit. Specifically this causes the Spek tests not to execute, when JUnit is used in the same test run. This can lead to false positives, and the only solution is to hold Spek tests separately.

It is hoped that version 2.0 of Spek will resolve this

26.6 Documentation

1. Documentation should be kept with and maintained at the same time as its associated code. This is not currently achieved because all the documentation is in the main Krail repository. Since the move to Sphinx / ReST, it may be achievable using an include
2. Javadoc (or Kotlin equivalent) does not need to state the obvious. But, if you think you would benefit from some notes when you come back to the code in 5 years' time - then write those notes.

CHAPTER 27

Bootstrap

In order to allow the same application code to run in both Vertx and Servlet environments, a bootstrap sequence is needed.

This was originally enables the Guice Injector to be held in a location suitable to the environment, so that it can be retrieved during *deserialisation*

It also sets the value of the @RuntimeEnvironment option, so that the application may make other adjustments if needed.

27.1 Injector Location

In a standard Servlet environment, a static variable is the simplest way to hold a reference to the Guice Injector. This is provided by `InjectorHolder`, but should usually only be accessed through `InjectorLocator` - this enables the code to be application portable, because the `InjectorLocator` implementation is environment specific.

In a Vertx environment, the *injector instance* is held in `Vertx.currentContext`. Again, to ensure portability, access should be through `InjectorLocator`

27.2 Guice Bindings

Environment specific bindings are defined in `ServletEnvironmentModule` and `VertxEnvironmentModule`, for:

- `InjectorLocator`
- `SerializationSupport`

27.3 Bootstrap file

This is described in the User Guide <../userguide/userguide-bootstrap>‘

27.4 Detecting the Environment

The *Bootstrap* process provides detection of the runtime environment, which can be accessed by Guice injection

Listing 1: Java

```
public class MyClass {

   @Inject
   protected MyClass(@RunningOn RuntimeEnvironment runtimeEnvironment){

   }
}
```

Listing 2: Kotlin

```
class MyClass @Inject constructor(@RunningOn runtimeEnvironment:RuntimeEnvironment)
```

CHAPTER 28

Configuration

28.1 Objective

As it says at the start of the Tutorial, the objective is to give you, the Krail developer, the best of both worlds - quick results, but still the freedom to modify things however you wish. Of course, as with any form of freedom, there is also responsibility. So if you break something, you can guess who will be expected to fix it !

28.2 Configuration levels

Configuration is possible at multiple levels, and how you use them is largely up to you.

28.2.1 Level 0 - Requires a re-compile

When making fundamental changes - for example using a different implementation for a specific interface - reconfiguration is through Guice modules. As these are in code, this will of course require a recompile. Guice annotations also play a part in this. Krail is programmed almost entirely to interfaces, so at this level of configuration you could change just about anything. Or, indeed, break just about anything. Also in this category is the use of the EventBus, since you can choose to accept or send messages and act upon them as required. You will see examples of this throughout the Tutorial.

28.2.2 Level 1 - Loadable configuration

Even in small applications, there are times when it is inconvenient to require a recompile in order to change system behaviour. Krail integrates Apache Commons Configuration, which provides support for many formats for loading configuration. Krail uses the extended properties file format itself (basically an ini file with sections). It is also possible to merge multiple inputs, thus supporting a modular approach if you build a Krail application with multiple libraries or modules. It is up to you to ensure the files are loaded when needed, but not unnecessarily often. There will be at least one example of this in the Tutorial.

28.2.3 Level 2 - Dynamic options

A further level of configuration is provided through the `Option` class. This enables the update of options as the application is running - it is up to the application to dynamically update if that is required. This configuration is also multi-layered, so that there is the potential, for example, to have options set at system, department and individual user level (Krail does not determine this structure of this hierarchy, as that it application dependent, it simply provides a mechanism to enable it). Option is typically used to allow users to make their own choices, but also provide typical defaults based on one or more hierarchies they are a member of. This is quite a large subject and therefore has its own section, *Option and UserHierarchy*.

CHAPTER 29

Event Bus

Krail uses a publish - subscribe Event Bus in place of the common Event Listener based Observer pattern. The Guava EventBus page gives a good summary of the reasons for choosing this approach (under One Minute Guide). In fact, the idea was originally to use the Guava implementation, but MBassador was chosen in its place because:

- it supports both weak and strong references (weak by default). Guava supports only strong references.
- it supports synchronous and asynchronous buses

29.1 Overview

Most of the work is done in the `EventBusModule`, which binds bus instances to match the *Krail scopes* of **@Singleton**, **@VaadinSessionScope** and **@UIScope**. You don't have to use all three, but there is a natural correlation between a bus and a scope.

The 3 implementations are represented by 3 binding annotations **@GlobalMessageBus**, **@SessionBus** and **@UIBus** respectively - each of which also is used to bind implementations for bus configuration, bus configuration error handlers and publication error handlers.

This means that by simple configuration changes in EventBusModule, you have 3 possible bus implementations matching Guice scopes, with each having individual configuration objects, individual error handlers if required and a choice between synchronous and asynchronous messaging.

The EventBusModule also uses Guice AOP to automatically subscribe classes annotated with @Listener.

The **@GlobalMessageBus** is asynchronous by default, as most of the publishers and subscribers are likely to be **@Singletons**, and therefore thread-safe. The other 2 buses are synchronous.

All of the Guice configuration can of course be changed by replacing / sub-classing EventBusModule and updating your `BindingManager` in the usual way.

29.2 Publishing Messages

Simply inject the **@GlobalMessageBus**, **@SessionBus** or **@UIBus** you want:

```
public class MyClassWithPublish {

private final eventBus;

    @Inject
    protected MyClassWithPublish(@SessionBus PubSubSupport eventBus){
        this.eventBus=eventBus;
    }

}
```

Use an existing `BusMessage` implementation, or create your own message class - which can be anything which implements the `BusMessage` interface. (There are no methods in the interface, it is there for type safety and to help identify message classes). At the appropriate point in your code, publish your message: ‘ public void someMethod(){ .... do stuff ... eventBus.publish (new MyBusMessage(this, someMoreInfo)); } ‘

29.3 Subscribing to Messages

Annotate the class with **@Listener** (which can also specify strong references)

Annotate the method within the listener class which will handle the message with **@Handler**. The method must have a single parameter of the type of message you want to receive

```
@Handler
public void MyMessageHandlerMethod(MyBusMessage busMessage){

    MyClassWithPublish sender = busMessage.getSender();

}
```

There are some other very useful features such as Filters and Priority .. see the MBassador documentation.

29.4 Automatic Subscription

During object instantiation, Guice AOP uses an InjectionListener in the `EventBusModule` to intercept all objects whose class is annotated with @Listener. The rules defining which bus to subscribe to are defined in an implementation of EventBusAutoSubscriber, which you can of course replace by binding a different implementation. The default implementation uses the @SubscribeTo to complement the association rules:

if a **@SubscribeTo** annotation is present, the buses defined by the annotation are used, and no others (Services are an exception, see below) if a **@SubscribeTo** annotation has no arguments, it is effectively the same as saying “do not subscribe to anything even though this class is marked with a @Listener” if there is no **@SubscribeTo** annotation a Singleton is subscribed to the **@GlobalMessageBus** anything else is subscribed to **@SessionBus** Note that **@Listener** and **@SubscribeTo** are inherited, so can be used on super-classes, but be overridden if re-defined in a sub-class.

29.5 Services and Messages

Service implementations make use of the Event Bus to automate starting / stopping and restarting interdependent services. Many Service implementations are **@Singletons** (though they do not have to be), so the **@GlobalMessageBus** is used and ALL Service objects are automatically subscribed though `AbstractService` to the **@GlobalBus**. It is probably unwise to change that.

CHAPTER 30

Functional Testing

For a functional test, exercised through the user interface, it is likely you would want to use one of the tools designed for just that purpose. Currently these would most likely be one of:

- Vaadin TestBench
- Selenide
- Selenium

Vaadin TestBench is obviously Vaadin aware, but has a licence cost. Selenide is free and open source. Both use Selenium underneath to remove some of the issues of testing in an AJAX environment.

30.1 Component Ids

All the above tools provide various methods of detecting an element within a web page. The most robust is to use a CSS Selector, which in Vaadin's case is provided by a `Component.id`

To assist testing, Krail automatically assigns a hierarchical id to selected components. This is done by an implementation of `ComponentIdGenerator`

This id is in the form of *MyView-component-nestedcomponent-nestedcomponent* to whatever depth is defined by your views and components. By default, anything which implements `Layout` is ignored, as these do not usually declare any components, and are not usually required for functional testing.

You can, however, use **@AssignComponentId** annotation to change this

30.1.1 Affect on Performance

Using CSS selectors makes robust testing through the UI a lot easier, but does have the penalty of incurring additional network traffic for all the extra labels, which you might not want in a production environment - though if performance is not an issue, they could also be used for application monitoring.

There is an outstanding issue to make it possible to switch this feature off via configuration.

30.2 Page Loading

One of the problems with automated testing is knowing when a page is ready to be tested. A `PageLoading` message is despatched on the `MessageBus` as the transition from one page to another is started, followed by a `Page Ready` message once the page has been built and data has been loaded.

In `SimpleUI`, the `PageLoadingMessage` sets the `NavigationBar` title to "Loading ...". When the `PageReadyMessage` is received, the title is set to the name of the View. This can be used by functional test code to determine whether the page is ready for testing.

30.3 Functional Test Support

VaadinTestBench has been replaced by Selenide for Functional Testing. This solution is not as complete as TestBench, but covers many use cases.

Component ids are now generated automatically to support functional testing.

A `FunctionalTestSupport` object provides a model of route to View / UI, and the components they contain.

To complement this, there is some early but useful work held currently in the test-app project which generates Page Objects for functional testing. These, along with some framework code, enable testing using Selenide, and could be extended easily for use with Vaadin TestBench - the objective it to enable the use of different test tools without changing the tests

See `GeneratorPageObjects` and the other classes in uk.q3c.krail.functest, in the test-app project

The code behind this will eventually become a separate library.

CHAPTER 31

Guice Scopes

This section considers Guice and in particular its relationship with Vaadin

To understand Guice itself, the Guice documentation is a good place to start. This documentation only addresses points which relate to its use in Krail

If you think you are not are not familiar with the idea of scopes, actually you probably are - at its simplest level, the principles are no different to thinking of variables having scope.

31.1 Vaadin Environment

The Vaadin architecture is significantly different to a typical Web environment. There are three scopes used by Krail to reflect Vaadin's design:

31.1.1 UI Scope

UI Scope represents a Vaadin UI instance, and is generally equivalent to a browser tab. To give a class this scope, apply the **@UIScoped** annotation to the class and instantiate with Guice.

31.1.2 Vaadin Session Scope

Vaadin session scope represents a `VaadinSession` and is generally equivalent to a browser instance. To give a class this scope, apply the **@VaadinSessionScoped** annotation to the class and instantiate with Guice.

31.1.3 Singleton

A Singleton has only one instance in the application. To give a class this scope, apply the **@Singleton** annotation to the class and instantiate with Guice.

Singleton classes must be thread safe.

31.2 AOP

Guice AOP is used by Krail, and if you are not familiar with it the main points to note are:

- Guice AOP works only on method interception
- It does **NOT** work on private, static or final methods - this is very easy to forget when stubbing methods with an IDE!
- For Guice AOP to work, Guice must instantiate the object

CHAPTER 32

I18N

THIS SECTION IS OUT OF DATE & REQUIRES REVIEW

32.1 Introduction

Early in the development of Krail it was decided to support I18N as an integral part of the development. Although many applications only need to support one language, trying to add internationalisation (I18N) later can be a major challenge. I18N requires the separation of literal strings into files for translation, but this actually makes good sense even if only one language is required, since it keeps a good separation between the use of messages and the exact wording of them. In addition, the need for parameterised messages will occur regardless of the number of languages supported - so we concluded that it makes sense to always write an application as if I18N will be required. And if one day your single language application suddenly has to go multilingual, the only thing required will be the translations.

32.2 The Basics

In the context of I18N, each piece of text needs a pattern, optionally with placeholders for variable values, and a key to look up that pattern for one or more Locales - then there needs to be something to bring it all together to find the right pattern for a selected Locale, fill in variable values and provide the result.

32.2.1 The Pattern

The pattern needs to be of the form:

```
The {1} task completed {0} iterations in {2} seconds
```

Different languages may require the parameters to be in a different order - the number in the {0} represents the order in which the values should be assigned, so for this example values of 5, “last”, 20 will become:

```
The last task completed 5 iterations in 20 seconds
```

32.2.2 The Key

A key in Krail is an enum. This has many advantages over the usual approach of using String constants, especially when combining modules which may need to define their own keys in isolation from each other. They are also more refactor-friendly. An I18N key class must implement the I18NKey interface:

```
public enum LabelKey implements I18NKey { Yes, No, Cancel }
```

A key class represents a "bundle".

32.2.3 The Bundle

The term "bundle" is used throughout native Java I18N support and Krail uses the term in a similar way. It represents an arbitrary set of keys and the collection of patterns, of potentially multiple languages, that go with the keys. An enum implementation I18NKey class therefore represents a set of keys for a bundle.

32.2.4 Bundle Reader

Patterns potentially come from different sources. Krail supports the property file system used by native Java. It also provides a class based implementation, and the Krail JPA module provides a database implementation. All of these - and others if required - implement a BundleReader interface to read a pattern from a file, class, database - perhaps a web service - or wherever the implementation is designed to work with.

32.2.5 Pattern Source

The `PatternSource` combines inputs from potentially multiple `BundleReader`s into one source. This is configurable through `I18NModule` to query the readers in whatever order is required - the first to return a value is used. If necessary a different order for each Bundle, so a database source could be the primary for one bundle and secondary for another.

32.2.6 Translate

The `Translate` class is the final step in bringing the pieces together. It looks up the pattern for a `Locale`, via the `PatternSource`, and combines that with the parameter values it is given. For the example above the call would be:

```
translate.from (MessageKey.Task_Completion, Locale.UK, 5, "last", 20)
```

If `Translate` cannot find a pattern, it will default to using the key name (with underscores replaced with spaces). This is useful when prototyping, as the pattern can still be meaningful even if not strictly accurate. That's why you will find many of the Krail examples break with the convention of using all uppercase for the I18NKey enum constants.

Note that if "last" also need to be translated, `Translate` will accept and perform a nested translation on an I18NKey (though the nested value cannot have parameters - if that is required, two calls to Translate will be needed)

```
translate.from (MessageKey.Task_Completion, Locale.UK, 5, LabelKey.last, 20)
```

You do not always have to specify the Locale - the default is to use CurrentLocale.

32.2.7 Current Locale

The `CurrentLocale` implementation holds the currently selected locale for the user. The default implementation checks things like the browser locale and user options to decide which locale to use. CurrentLocale can be injected anywhere it is required, and the Translate class will use it if no specific Locale is supplied when calling the from() method.

32.2.8 Configuration

A number of things can be configured in the `I18NModule`, part of the Guice based configuration - it is worth checking the javadoc for this. Some configuration is also available via `Option`.

32.3 Managing Keys

To make it just a little easier to find values in what can be a long list, the Krail core uses 3 enum classes to define message patterns:

- Labels : short, usually one or two words, no parameters, generally used as captions
- Descriptions : longer, typically several words, no parameters, generally used in tooltips
- Messages : contains parameter(s).

Note that this is simply a convention - you can call them whatever you wish.

For each there is enum lookup key class:

- LabelKey,
- DescriptionKey,
- MessageKey.

For a class implementation there needs also to be a corresponding map of value (default names of Labels, Descriptions and Messages) extended from `EnumResourceBundle`. For a property file implementation there needs to be a file (or set of files for different languages)

Using enums as I18N keys has some advantages, particularly for type checking and refactoring - but it also has a disadvantage. Enums cannot be extended. To provide your own keys (which you will unless you only use those provided by Krail) you will need to define your own I18NKey implementation, as described in the Tutorial - Extending I18N.

32.4 Managing Locale

32.4.1 CurrentLocale

`` `CurrentLocale` `` holds the locale setting for the current `` `VaadinSession` ``. Once a user has logged in, it is also possible to set the locale for a specific component, using the annotations described below.

32.4.2 Using I18N with Components

A typical component will need a caption, description (tooltip) and potentially a value. These need to be set in a way which recognises the correct locale, and potentially to update if a change of locale occurs.

@Caption

The **@Caption** annotation marks a component as requiring translation, and can provide caption and description

```
@Caption(caption=LabelKey.Yes, description=DescriptionKey.Confirm_Ok)
```

The application UI invokes the *I18NProcessor* to perform the translation during initialisation of any components it contains directly. When a view becomes current, its components are also scanned for *@18N* annotations and translated. *I18NProcessor* also updates the component's locale, so that values are displayed in the correct format.

When `CurrentLocale` is changed, any UIs associated with the same VaadinSession are informed, and they each update their own components, and their current view. When a view is changed, if the current locale is different to that previously used by the view, then the View and its components are updated with the correct translation.

When a field or class is annotated with *@I18N*, the scan drills down to check for more annotations, unless the annotation is on a core Vaadin component (something with a class name starting with 'com.Vaadin') - these clearly cannot contain I18N annotations. and therefore no drill down occurs.

@Description

Similar to *@Caption*, but without the caption !

@Value

Usually, it is the caption and description which would be subject to internationalisation, but there are occasions when it is a component's value which should be handled this way - a *Label* is commonly an example of this. Because the use of value is a little inconsistent in this context it has its own annotation.

Multiple annotations

You can apply multiple annotations - but note that if you define the locale differently in the two annotations, the result is indeterminate (that is, it could be either of the two locales that have been set).

Composite Components and Containers

There are occasions when an object contains components, and may not be a component itself, or possibly just not need translation.

For example, you have a composite component `MyComposite` which itself does not need a caption or description - but it contains components which do. For these cases, simply annotate it with @18N without any parameters, and `I18NProcessor` will scan `MyComposite` for any fields which need processing.

If `` `MyComposite `` is intended to be re-usable, it would probably be better to annotate the class with *@I18N*, so that it does not need to be annotated each time it is used.

32.4.3 Extending I18N

Annotation parameters cannot be generics, so will need to provide your own equivalent of *@Caption*, *@Description* and *@Value* to use your keys for annotating components for translation. The method for doing this is described in the Tutorial - Extending I18N.

32.4.4 Validation

The messages used in validation can be supported in the same way .. see the Validation section for details.

CHAPTER 33

Options and Hierarchies

The idea of providing users with options is a standard requirement for many applications, whether it is just letting them decide what they see on a page, or maybe the news feed they get. Krail provides an implementation which should be flexible enough for any application, with a minimum of effort. This guide describes the structure and principles behind Options - for detail of how to use them, please refer to the *Tutorial*.

33.1 Relationship to Configuration

Krail sees `Option` as the final layer of configuration. In practice, what matters is that the Krail developer has a huge amount of flexibility and control in managing configuration, including users' individual options.

33.2 Layers of Options

At its simplest, a user should be able to select and save options, then retrieve them from persistence next time they use the system. The user may not have used the system before, though, so we need some defaults to start with. In Krail these defaults are provided in code, as the `Option.get()` method requires a default value - this also ensures that behaviour is predictable if option values are missing.

So we have a user defined value for an option and a coded default. But now suppose we think it would be better if we could change some options for all users - "system options" in effect. Or to make things a bit more complicated, we want to set some options at system level, and allow users to override just some of them.

This is nothing more than a simple hierarchy, represented in Krail by `UserHierarchy`. If we simply say that values at the user level override those at system level, then we almost have what we want. And only allowing authorised users to change some of the `Option values`, those become system level options. So for this simple, 2 level hierarchy, the logic for retrieving an option value is quite simply to take the first non-null value we find from the following order:

- user level
- system level
- coded default

33.3 Controlling the Options

Of course, you don't have to give all users the facility to change all options - you may restrict changing some options values, for example, to sys admins, to provide consistency across the whole system.

Accessing options is always through the `Option` interface. This enables a simple, consistent API for storing and retrieving options.

33.4 Hierarchies

What has been described above is a simple, 2 layer, `UserHierarchy` implementation, and this is the default provided by Krail. But that may not be enough for you. Perhaps you are developing an application for a large, complex organisation, and what you would really like to do is have layers like those described above, but structured by geography or company structure - or maybe both.

That is easily achievable with your own implementation of `OptionLayerDefinition`. This interface has a method which returns a list (an ordered hierarchy) based on parameters of user and hierarchy name. For a specific user this may return "London, Europe" for geography, and "Engineering, Automotive, Off Road" for company structure - the data for these would probably both be obtained from another corporate system.

The principles described above remain the same, however - so for this example, `Option` will return the first non-null value found from a location hierarchy of:

- user
- city
- continent
- system
- coded default

There can be up to 98 layers between user and system levels, though we can think of no sane reason for wanting that many.

33.5 Storing the Options

None of this is of any use unless the option values can be stored. As with the rest of Krail, an interface, `OptionStore`, and a default implementation are provided. In this case, the default implementation is not very useful, as it only store the options in memory. A persistent version is planned, but in the meantime you could provide your own persistent implementation and bind it through a sub-class of OptionModule.

33.6 OptionKey

CHAPTER 34

Page Navigation

The Vaadin UI classes, Views and Navigator are all closely involved in the process of navigation. The Vaadin documentation gives a good description of how it works, but Krail brings some additional features to it.

In brief, the UI class represents a browser tab, and a View is placed within the UI. The View is then changed in response to navigation changes. You can have multiple UI classes, and multiple View classes. In Krail, the selection of which View to display is derived from the URI.

34.1 When to use a UI or View

One question which arises quite quickly is what should be part of a UI, and what should be in a View. For example a header bar could go in either. Our inclination is to use a UI containing only elements which will always appear on every page. Most of the user interface is then provided through the View.

Krail makes the use of Views even easier, and as a result probably makes the use of the UI class to hold user interface components less useful.

34.2 URI and Route

A central part of the way navigation works in Krail is the interpretation of the *URI*. The default implementation of `URIFragmentHandler`, is `StrictURIFragmentHandler`.

This provides a more strict interpretation of the UriFragment than Vaadin does by default. It requires that the URI structure is of the form:

```
http://example.com/domain#!finance/report/risk/id=1223/year=2012
```

This can be with or without the bang after the hash, depending on the `useBang` setting

where:

```
finance/report/risk/
```

is a "route" and is represented by a `KrailView`

and everything after it is paired parameters. If a segment within the paired parameters is malformed, it is ignored, and when the URI is reconstructed, will disappear. So for example:

```
http://example.com/domain#!finance/report/risk/id=1223/year2012
```

would be treated as:

```
http://example.com/domain#!finance/report/risk/id=1223
```

The year parameter has been dropped because it has no "="

Optionally you can use hash(#) or hashBang(#!). Some people get excited about hashbangs. Try Googling it

CHAPTER 35

Push

35.1 Background

The Vaadin-provided 'push to browser' mechanism uses Atmosphere, and this proved to be challenging for the author of the vertx-vaadin library, which Krail uses to run on Vertx.

Vertx also provides a 'push to browser' facility, but one which is an integral part of the Vertx Event Bus, with much greater functionality. In the words of the Vertx documentation:

```
The event bus forms a distributed peer-to-peer messaging system spanning multiple␣
↪server nodes and multiple browsers.
```

35.2 Krail, Push and Vertx

For good reasons, therefore, vertx-vaadin uses the Vertx push mechanism. In order to accommodate that, some changes are needed for Krail.

The push connection is managed by the Vaadin UI (ScopedUI in Krail), with an embedded helper implementation of `PushConfiguration`. The simple task of using a different connection (`SockJSConnection` for Vertx, `AtmospherePushConnection` for Servlet environments), is made complicated by the closed nature of the Vaadin code structure.

There are two places which need the correct connection to be set, as described in the related issue, namely:

- `ScopedUI` constructor or `init` method
- the `PushConfiguration.setPushMode()` method

The first is perfectly simple. The second, however, causes problems.

- the `PushConfigurationImpl.setPushMode()` method constructs and sets the connection using *new AtmospherePushConnection()* - this would mean that disabling and then re-enabling would switch back to the Atmosphere connector.

- The default implementation of `PushConfiguration`, `PushConfigurationImpl` is constructed in the declaration of the `pushConfiguration` field of UI
- the `pushConfiguration` field of UI is private and has no setter

35.3 Adaptations

Various methods of getting round these restrictions were considered, and all have their pros and cons. The simplest, if rather nasty hack, of replacing the default `PushConfiguration` by reflection was reluctantly considered the better option rather than duplicating a lot of the native Vaadin code. This is done by calling `overridePushConfiguration()` in the `ScopedUI` constructor.

This change is complemented by a new `PushConfiguration` implementation , `KrailPushConfiguration`, which is a direct lift of Vaadin code with the `setPushMode()` method changed to allow the construction of the correct push connection.

35.4 Detecting the Environment

`KrailPushConfiguration` needs to know which environment (Servlet or Vertx) it is running in. The Bootstrap process provides detection of the runtime environment, which is accessed by Guice injection.

CHAPTER 36

Persistence

36.1 Introduction

There are currently two forms of persistence made available in Krail:

- the krail-jpa module, and
- the "In Memory" classes - which are not strictly speaking persistent, but offer the same API to aid a fast development start up and some testability.

This section provides guidance on what a persistence implementation should provide to the Krail core. Both the In Memory and JPA implementations can be viewed as a way of understanding how this works.

For ease of description, the persistence provider here is unimaginatively called "XXXX" - although that may mean something to Australian readers.

36.2 Terminology

Throughout this section the terms "Persistence Unit" is used in the manner defined by JPA.

36.3 Identity

Very often an application will use a single persistence unit. However, this should not become a constraint, as other applications require multiple persistence untis. Krail should therefore enable the selection and use of multiple persistence units, if that is what the application requires.

Each Persistence Unit, and is associated services, must therefore be identifiable by an Annotation. The Annotation itself currently has no specific requirements.

36.4 Multiple Persistence Units from the same provider

Ideally, the persistence provider will support multiple persistence units for the same source type - for example, the krail-jpa module supports multiple PUs, each identified by their own unique annotation. The In Memory persistence provided by Krail, however, offers only a single PU, although that is identified by an annotation as required above. At the time of writing, an OrientDb library is being considered, which would only provide a single PU - but this would still be required to carry an annotation as required above.

36.5 Option

Krail core uses the `Option` class extensively, and by default, `Option` values are stored in an "in memory", volatile store. A persistence provider must provide support for `Option`, accessible to both the Krail developer's application and the Krail core.

`Option` requires a DAO implementation to read and set `Option` values. To support the presentation of `Option` values to the end user, an implementation of `OptionContainerProvider` is required. The Guice Module used to configure the PU must also provide a fluent method for the *BindingManager*' to enable the `Option` support for the PU.

The detailed requirements are therefore:

1. `XXXXOptionDao` which **extends** the `OptionDao` interface, with a Guice binding to the Identity annotation
2. `DefaultXXXXOptionDao` as the default implementation of `XXXXOptionDao`
3. The binding of `XXXXOptionDao` to `DefaultXXXXOptionDao` must be available to the Krail developer to override
4. `XXXXOptionContainerProvider` which **extends** the `OptionContainerProvider` interface, with a Guice binding to the Identity annotation
5. `DefaultXXXXOptionContainerProvider` as the default implementation for `XXXXOptionContainerProvider`
6. The binding of `XXXXOptionContainerProvider`to `DefaultXXXXOptionContainerProvider` must be available to the Krail developer to override
7. The Guice module should be called `XXXXModule`
8. The `XXXXModule` must provide a a fluent method `provideOptionDao()` which will create all the bindings listed above.
9. If the `provideOptionDao()` method for a PU is not invoked before the Guice Injector is created, the bindings listed above should NOT be created

36.5.1 Testing Bindings

The bindings for a PU enabled by invoking `provideOptionDao()` should return instances as defined below:

@XXXX1 OptionDao should return DefaultXXXXOptionDao

36.6 Pattern

The Krail core uses I18N patterns extensively, and by default are read from `EnumResourceBundle` instances. A persistence provider must provide support for reading and writing I18N patterns. Support for writing is required in

order to enable the copying of I18N patterns, and the provision of translations from within the Krail application.

36.7 EntityProvider or EntityManagerProvider

36.8 Generic DAO

CHAPTER 37

Serialisation

A Krail application would originally have only required to support serialisation in a high availability (HA) environment. In general, using sticky sessions in a clustered environment would be sufficient.

In addition to HA, running a Krail application on Vert.x places a further requirement for serialisation support. This introduces all the usual Java serialisation issues of non-Serializable classes.

Specifically, Vaadin is designed in such a way that it holds the entire UI state to memory, and therefore needs to serialise it to session when HA or other circumstances need to move a session.

37.1 Scope of impact

Given that the entire UI state needs to be serialised, this includes anything which implements the `UI` or `KrailView`, plus, of course, anything which they in turn require.

Even though components themselves should be `Serializable`, the largest impact is on views. The emphasis in these notes is on views, but applies equally to `UI` implementations.

The design pattern for Krail has been to use Guice constructor injection of many and varied dependencies into implementations of `KrailView`. More of these dependencies could undoubtedly implement `Serializable`, but there will always be cases where this is not possible.

The use of Guice introduces a further challenge - in order to be certain that dependencies are resolved consistently, Guice should be used to re-construct any transient dependencies after deserialisation.

37.2 Objectives

1. Use Guice to re-construct any transient dependencies after deserialisation
2. Must allow for transients which are either reconstructed by Guice or by developer code (a @NotGuice annotation?)
3. Make the process as simple and clear as possible for Krail application developers

4. If possible, make transition to a non-native serialisation process an easy option.
5. Allow Krail developers to populate other transient fields as they need to, before and/or after the injections

37.3 Options and Obstacles

In order to meet objective 1), any potential resolution requires that the Guice Injector is available during deserialisation.

37.3.1 Use of Injector.injectMembers

This would be very easy to implement in a readObject - a simply call (hooks for the developer to use before and after the injections have been ignored for this example):

```
private void readObject(ObjectInputStream in) throws ClassNotFoundException,␣
↪IOException {
    in.defaultReadObject();
    getInjector().injectMembers(this)
}
```

Obstacles

- This will only work with field or method injection - any non-serialisable fields would have to be field or method injected. While this would work, it imposes a restriction on constructor injection.
- This would mean abandoning or modifying constructor injection for all the current `KrailView` implementations
- Field injection has its own limitations, and most see it as a less attractive option. Difficulty of testing is usually overstated especially with the introduction of Bound Fields, and the choices are discussed in the Guice documentation.
- Even if Field injection were considered a good option, it would remove the choice from a Krail application developer.

37.3.2 Proxy serialisation

An old post the author made a long time ago, may also provide an answer. The relevant part is copied below:

> If you can hook the objects that you want serialized by adding a writeReplace() method, then you can use a serializable proxy to ship each over the wire. The proxy would contain the Key corresponding to the binding of the object. The trick here is establishing the reverse mapping from { object => key }, but it's possible through use of either the provision API or plain java code (I've done the latter myself).
>
> The proxy would have a readResolve() method that can find a handle to the injector on the JVM (several options exist for how to do this), and then it would return "injector.getInstance(key)." This solution would allow you to have non-serializable types as "private final transient" instance vars on a serializable object. Looks strange, but it's possible.
>
> As another solution, if you could only hook the deserialization code, you could then declare the non-serializable types as non-final and use member injection when types are deserialized (via "injector.injectMembers(deserializedObject)").

Obstacles

- `Key` is not serialisable, and nor is TypeLiteral, which would be an alternative. Guava TypeToken may be an option
- Defining a proxy would need to cater for generics, which is complicated by `Key` not being serialisable
- Reflection is still required
- Fields would have to be annotated with binding annotations where there are two instances of the same type being injected - there is no other way to match a constructor parameter to the field it is assigned to.
- If we can serialise a representation of the key on write, we could just as easily construct the key on read, and just inject an instance from that Key

37.3.3 Bespoke transient field initialiser

It would seem possible to create a routine to populate transient fields by reflection, using an Injector, as part of readObject, in `ViewBase` This might mean repeating the binding annotations on fields (but without the @Inject), where there are multiple injections of the same type.

Obstacles

- Fields would have to be annotated with binding annotations where there are two instances of the same type being injected - there is no other way to match a constructor parameter to the field it is assigned to.

37.4 Conclusion

Of these choices, the *Bespoke transient field initialiser* seems to offer the best solution, in that it all it requires is that the Krail developer annotates transient fields with binding annotations where needed.

CHAPTER 38

Services

38.1 Managing the Lifecycle

38.1.1 State Changes and Causes

The following table summarises the changes of state and cause, depending on the old state and the call made. If any call is made that does not appear in these tables, or listed under **Ignored Calls**, an exception is thrown.

No errors during call

This table assumes no errors occur during the call

old state	full call	short call	new state & cause
INITIAL	start(STARTED)	start()	RUNNING, STARTED
RUNNING	stop(STOPPED)	stop()	STOPPED, STOPPED
RUNNING	stop(FAILED)	fail()	FAILED, FAILED
RUNNING	stop(DEPENDENCY_STOPPED)	dependencyStop()	STOPPED, DEPENDENCY_STOPPED
RUNNING	stop(DEPENDENCY_FAILED)	dependencyFail()	STOPPED, DEPENDENCY_FAILED
STOPPED	start(STARTED)	start()	RUNNING, STARTED
FAILED		reset()	INITIAL, RESET

Error occurs during call

This table assumes that an error occurs during the call

old state	full call	short call	new state & cause
INITIAL	start(STARTED)	start()	FAILED, FAILED_TO_START*
STOPPED	start(STARTED)	start()	FAILED, FAILED_TO_START*
INITIAL	start(STARTED)	start()	INITIAL, DEPENDENCY_FAILED **
STOPPED	start(STARTED)	start()	STOPPED, DEPENDENCY_FAILED **
RUN-NING	stop(STOPPED)	stop()	FAILED, FAILED_TO_STOP
RUN-NING	stop(FAILED)	fail()	FAILED, FAILED
RUN-NING	stop(DEPENDENCY_S TOPPED)	dependen-cyStop()	FAILED, FAILED_TO_STOP
RUN-NING	stop(DEPENDENCY_F AILED)	dependencyFail()	FAILED, FAILED_TO_STOP
FAILED		reset()	FAILED, FAILED_TO_RESET

- Error in the Service that was called
  - Error in a dependency of the Service that was called

Ignored Calls

A call to start() when the state is STARTING or RUNNING is ignored A call to stop(), fail(), dependencyStop(), dependencyFail() when the state is INITIAL, RESETTING, STOPPED or FAILED is ignored A call to reset() when the state is INITIAL or RESETTING is ignored

38.2 Service Instantiation

When a Service is instantiated through Guice, AOP in the ServicesModule calls ServicesModel.addService, which also creates all its dependencies from the 'template' provided by the ServicesModel.classGraph

This means that the instanceGraph should always have a complete set of dependencies for any Service instantiated through Guice.

CHAPTER 39

Testing

39.1 Introduction

This pages captures some tips and techniques to assist you in testing your application.

39.2 ResourceUtils

The ResourceUtils class is part of the core, and is used to look up various directories, and can be used to manipulate the environment when testing. In the VaadinService example you can see that it is used to retrieve the user's home directory, but what is not immediately obvious is that is also used to determine the application base directory and configuration directory, and these are derived from the VaadinService. If you have mocked the VaadinService, as described, then you can set up application configuration however you wish for testing.

39.3 VaadinService

You will often find that your test needs a VaadinService to run, but of course is not usually available in a test environment - unless you are running full functional testing. To overcome this, we mock the service, with the help of ResourceUtils like this:

```
static VaadinService vaadinService;

@BeforeClass
public static void setupClass() {
   vaadinService = mock(VaadinService.class);
    when(vaadinService.getBaseDirectory()).thenReturn(ResourceUtils.
↪userTempDirectory());
    VaadinService.setCurrent(vaadinService);
}
```

CHAPTER 40

User Access Control

So what actually happens?

Krail has a `MasterSitemap`, which contains all the page definitions for the whole site. This is built from the page definitions you provide using either the direct method or annotation method you covered in Tutorial - Pages and Navigation.

When a user logs in, the `MasterSitemap` is copied to a user-specific instance of `UserSitemap`. However, only those pages which the user is authorised to see are actually copied across, and displayed in the navigation components. This means that either the pages must be public, or the user must have permissions to see them in order for them to be displayed.

During the process of copying from the `MasterSitemap` to the UserSitemap, each page is checked to see whether the user has permission to view it - if not, then it is not copied to the `UserSitemap`. This provides one layer of security, and it also means that any attempt by a user to access a url not in the UserSitemap is rejected with a "page does not exist" message, not a "page is not authorised".

CHAPTER 41

Validation

41.1 Introduction

Krail uses the Apache BVal implementation of JSR303 to provide validation. It also integrates Apache BVal with the Krail I18N framework, so that all I18N requirements can be managed through the same process. These are some of the things you may want to do with validation.

Validation is invoked automatically through Krail's implementation of `BeanFieldGroup` (basically a form without the layout), once its fields have been annotated.

41.2 Use standard javax Validation

That's easy. Just use it the same way as you always do. This also true of the additional constraints Bval provides (**@NotEmpty** and **@Email**)

41.3 Use a different message for a single use of a javax annotation

If you want to change a message in just one or two places, for example, to change a validation failure of **@Min** to say: "speed really, really must be less than 20" you could use standard javax and provide a message pattern: ' @Min (value=20, message="{0} really, really must be less than {1}" private int speed; ' This is ok for one language but it will not translate. A better option might be: ' @Min (value=20, message="{com.example.ValidationKey.MinReally}") private int speed; ' Note the "{}" around the message, this denotes a message key rather than a pattern. Note also that this must be a valid `I18NKey`.

This method means you can take advantage of Krail's translation mechanism, but you do lose the type-safety Krail normally provides by using enum keys. There are no alternatives because of Java's limitations on what can be declared in an annotation.

41.4 Change a javax message for all uses

If you want to change the message for all uses, there is a facility within the Bval implementation to do that. Krail provides a method in `KrailValidationModule` to assist.

```
public class MyValidationModule extends KrailValidationModule{

[source]
----
@Override
protected void define() {
    addJavaxValidationSubstitute(Min.class,com.example.ValidationKey.Min);
}
----

}
```

41.5 Move all translations to one source

You may wish to put all your translations into one place, rather than have the validation translations held separately. There could be many good reasons for doing so, and there is an open ticket to provide a utility to migrate the standard keys and patterns to the I18N source of your choice. You will need to provide a set of I18NKeys for the validation messages (the full set of keys used by Apache Bval are listed below). Then, by using the substitution method shown above, all standard javax.validation.constraints and org.apache.bval.constraints messages can directed to use the new Krail keys.

41.5.1 standard

javax.validation.constraints.Null.message=must be null javax.validation.constraints.NotNull.message=may not be null javax.validation.constraints.AssertTrue.message=must be true javax.validation.constraints.AssertFalse.message=must be false javax.validation.constraints.Min.message=must be greater than or equal to {value} javax.validation.constraints.Max.message=must be less than or equal to {value} javax.validation.constraints.Size.message=size must be between {min} and {max} javax.validation.constraints.Digits.message=numeric value out of bounds (<{integer} digits>.<{fraction} digits> expected) javax.validation.constraints.Past.message=must be a past date javax.validation.constraints.Future.message=must be a future date javax.validation.constraints.Pattern.message=must match the following regular expression: {regexp} javax.validation.constraints.DecimalMax.message=must be less than or equal to {value} javax.validation.constraints.DecimalMin.message=must be greater than or equal to {value}

41.5.2 additional built-ins

org.apache.bval.constraints.NotEmpty.message=may not be empty org.apache.bval.constraints.Email.message=not a well-formed email address

41.6 Create a Custom Validation

There are many cases where a custom validator can be useful, and Apache Bval does enable the creation of custom validators. With one small addition, a custom validation can also integrate neatly with Krail's enum based I18N. There are three parts to the creation of a custom validator - the validator itself, the annotation used to invoke it and the key for

the message. Most of this is standard JSR303 - the only difference in the annotation is the `messageKey()` method needed to enable the use of Krail I18N keys.

41.6.1 The annotation

```
import javax.validation.Constraint;
import javax.validation.Payload;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import com.example.ValidationKey;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {AdultValidator.class})
public @interface Adult  {
    ValidationKey messageKey() default ValidationKey.Must_be_an_Adult;

    String message() default "krail";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default {};

    long value() default 18;

    /**
     * Defines several <code>@Adult</code> annotations on the same element
     * @see @Adult
     *
     *
     */
    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        Adult[] value();
    }
}
```

41.6.2 The Constraint Validator

```
public class AdultValidator implements ConstraintValidator<Adult, Number> {

    private long minValue;

    public void initialize(Adult annotation) {
        this.minValue = annotation.value();
    }
```

(continues on next page)

(continued from previous page)

```
    public boolean isValid(Number value, ConstraintValidatorContext context) {
        if (value == null) {
            return true;
        } else if (value instanceof BigDecimal) {
            return ((BigDecimal) value).compareTo(BigDecimal.valueOf(minValue)) != -1;
        } else if (value instanceof BigInteger) {
            return ((BigInteger) value).compareTo(BigInteger.valueOf(minValue)) != -1;
        } else {
            return value.longValue() >= minValue;
        }

    }
}
```

41.6.3 The Key

```
public enum ValidationKey implements I18NKey {
    Too_Big, Must_be_an_Adult
}
```

CHAPTER 42

Vertx

There are a couple of libraries on Github which provide Guice integration with Vert.x, but don't meet Krail's requirements.

42.1 Injector scope

It would be entirely possible to have Singleton scope be equivalent to Vert.x instance scope. At first sight, this looks attractive, but it would promote data sharing in a distributed application.

This would reduce the isolation / encapsulation between services and is therefore considered to be a "bad thing".

The Krail implementation considers a Verticle as Singleton scope (the Injector is created in the Verticle), encouraging the use of asynchronous events rather than sharing data.

CHAPTER 43

License

Krail is licensed under Apache 2.0

CHAPTER 44

Glossary

Fragment See URI

Route See URI

Sitemap The Krail Sitemap describes, as you would expect, the structure of the application. However, it is not just a passive output from a site, but an integral part of the application design - it brings together a route, its associated View and an I18N key for translating the page title.

View A View is almost as described in the Vaadin handbook - the only difference with a `KrailView`, as opposed to a standard Vaadin View, is that is modified to work with Krail's Guice enabled navigation.

URI Of course there is only one correct definition of 'URI', but in a Krail context it is the way the structure of the URI is interpreted which becomes important. This interpretation is defined by an implementation of `URIFragmentHandler`, and Krail's default implementation is `StrctURIFragmentHandler`. See the javadoc for that class for a definition of how it separates 'pages' from parameters. As Krail has evolved, the terminology used to describe various elements of a URI has become a bit confused. This section sets out how it should be - but at the moment, other documentation (and method / field naming) are inconsistent. Hopefully the planned move to Vert.x will not change anything further

These terms assume the use of `StrictURIFragmentHandler`

By example:

URI:: com.example.myapp/#members/detail/id=1 - the whole thing

baseUri:: com.example.myapp/

fragment:: members/detail/id=1

route :: members/detail

parameters:: id=1

Index