JHipster I18N App Tutorial

Contents

Version	Author	Purpose	Date
1.0	Samuel Huang	Initial Draft for I18N app Tutorial	13-July-2016

Introduction

This document is a walkthrough of my **I18N** web application implemented using **JHipster**. I tried to document most of the problems encountered, fixed and cool things learned along the way, as a result, this document could also serve as an informal tutorial for JHipster.

Before implementing the I18N app, I wasn't familiar with **AngularJS** despite all the Buzz I read about. So what better way to learn a new Javascript web app framework than to write a sample web app using it? Out of this came the I18N application.

For someone new to JHipster and wanting to use AngularJS with Java, you are in for a treat as it comes with a lot of state of the art open source tools bundled together to make the life of a Java web application developer much easier when starting out a new web app.

Not to mention all the cool things one could pick up along the way, which is arguably the most important reason of using it in the first place. Just think about Spring Boot, Spring Data JPA, Spring Rest, Hibernate, AngularJS, AngularUI, JQuery, Bootstrap, LibSass, Gradle/Maven, Liquibase, Gulp, NPM, Node, Bower, Proctractor, Gatling (gun), Elastic search, JWT, … (too many buzzwords) all come together and set up to run nicely with each other. You get the picture.

Probably the number 1 complain or pain of using JHipster will be the staggering amount of tools one will need to learn initially (and getting stuck) before becoming efficient with it. The JHipster communities (e.g. StackOverflow) are very active. Ask and you shall receive. The time and effort spent will be worthwhile once the learning curve is over. Hopefully, this tutorial will make the journey of climbing over the learning curve a smooth one. After all, this is a **Hipster** framework!

Other Excellent JHipster Tutorials

Introducing JHipster

JHipster Mini Book

Official JHipster website

Who this document is for

A Java Hipster

Tutorial Style

The journey of developing I18N app is documented in a blog like fashion.

To stop re-inventing the wheels, I will strive to use links to point readers to other web sites for detailed explanation. So no go over the basics of how to set up and run JHipster. A lot of screen shots will be used.

Often a problem encountered or feature that needs to be implemented will be described with screenshot(s). Then some link(s) or sample code will briefly explain how it's resolved. Screenshot(s) may be included to show the outcome. Readers will then be referred to Git commits to check out the details.

This tutorial is by no means claiming to use the best practice even though I tried my best. Feedback is welcome.

Code from tutorial

All the code and Git commits referred to in tutorial is available at https://github.com/sam888/jhipster_i18n/.

Whenever the Git icon ♦ appears, it indicates a reference to a Git commit. Reader can check that commit for the code discussed in that section.

How to run I18N app
See how-to-run.txt from the downloaded project at the link above.

About the Author
Samuel Huang. A hipster Java developer currently living in Auckland, New Zealand

What is I18N app and its motivations

In short, the I18N sample application I am going to implement will provide Internationalization by database as opposed to static JSON files. Still confused?

By default, JHipster generated application uses static JSON files for doing internationalization (i.e. I18N for short). Think of those JSON files as the properties files containing key-value pairs to provide customized labels per client basis in JEE web applications.

For example, visit the link *http://localhost:9000/i18n/en/global.json* in your running JHipster app will show the labels from *global.json* as

```
{
  "global": {
    "title": "Jhipster_i18n",
    "browsehappy": "You are using an <strong>outdated</strong> browser. Please <a
href=\"http://browsehappy.com/?locale=en\">upgrade your browser</a> to improve your experience.",
    "menu": {
      "home": "Home",
      "jhipster-needle-menu-add-element": "JHipster will add additional menu entries here (do not
translate!)",
      "entities": {
        "main": "Entities",
        "locale": "Locale",
        "module": "Module",
        "resourceBundle": "Resource Bundle",
        "keyValue": "Key Value",
        "jhipster-needle-menu-add-entry": "JHipster will add additional entities here (do not translate!)"
      },
      ......
  "error": {
    "server.not.reachable": "Server not reachable",
    "url.not.found": "Not found",
    "NotNull": "Field {{ fieldName }} cannot be empty!",
    "Size": "Field {{ fieldName }} does not meet min/max size requirements!",
    "userexists": "Login name already used!",
    "emailexists": "E-mail is already in use!",
    "idexists": "A new {{ entityName }} cannot already have an ID"
  },
  "footer": "This is your footer"
}
```

AngularJS would access the label with key *global.title* from above in html file with something like

> *Hipster*

and replaces the content of with the actual value of label (which is Jhipster_i18n from global.json) after parsing so it ends up as: *Jhipster_i18n*

So what's the motivation for providing I18N by database as opposed to static JSON files? If something is not broken, don't fix it right?

Imagine a web app used by multiple clients where each client requires its own set of customized labels. You can see where this is going now. Maintaining the labels can start to get complicated, especially if the client demands to change a label on the fly on their server(s).

To change a label would require modifying the corresponding JSON file then deploy it to the server again. This could potentially be an error prone and time consuming process (company politics & Change Control process). Now if the labels were stored in database, served by web app through a restful service, customizable in web app UI and see changes in real time, one can then start to see how useful this can be.

To top it off, once a request has been made to access the JSON data at a specified URL, that data could be cached on the server side so the 2nd call to access it won't result in a call to the database again, instead the JSON data stored in the cache will be served to speed up performance. And all these could be implemented in the backend (i.e. Java side) without changing anything in the frontend AngularJS.

This pretty much sums up the functional design of what I18N app is supposed to do.

Functional Design

- Strictly speaking, internationalization will need to consider Language code, Country
- code and a variant field (e.g. dialect) for the definition of Locale. But for simplicity, the I18N app will only use Language code to represent a Locale and serve request for JSON data(containing the labels) with the URL format: *http://localhost:9000/i18n/{language-code}/{module-name}.json.*

- I18N app will serve requests to access all I18n JSON files (each has a different URL) with a restful service

- Each I18N JSON file will be represented by a ResourceBundle record

- Each ResourceBundle record will have a list of labels, each of which will be represented as a KeyValue record. This is the classic one-to-many (parent to child) relationship mapping

- I18N app will also have UI to create/edit/delete ResourceBundle and KeyValue records

- Looking at the URL *http://localhost:9000/i18n/en/global.json* and others for accessing JSON files, we can see the pattern *http://localhost:9000/i18n/{language-code}/{module-name}.json.*

  Each unique combination of {language -code} and {module-name} would require its own ResourceBundle record => Each {language -code} or {module-name} is just a meta-data record that can be mapped to multiple ResourceBundle records.

  So a Locale entity needs to be created to represent {language -code} with *one-to-many* mapping to ResourceBundle.

A Module entity can be created to represent {module-name} with *one-to-many* mapping to ResourceBundle.

- The requirement resulted from previous one. A resource bundle entity will need to have unique constraints of Locale and Module entities, i.e. no two resource bundle records can have the same Locale & Module

- Also the name property of resource bundle has to be unique. Doesn't make sense to have two resource bundles having the same name => unique constraint on name column.

- I18N app will have UI to create/edit/delete Module and Locale records described above

- An extra *status* field in ResourceBundle table to allow switching of static JSON file to database driven JSON data. This should ease transition of static JSON file to database driven JSON data.
  When the status is 'DISABLED', nothing will be returned to the request for JSON data.

  When the status is 'STATIC_JSON', static JSON file will be used to serve the request. This basically behaves the same before any change.

  When the status is 'DATABASE_JSON', JSON data will be retrieved by the database to serve the request by using a restful service.

Database Design

The functional design above will lead to the following database design.

Generated by SchemaSpy
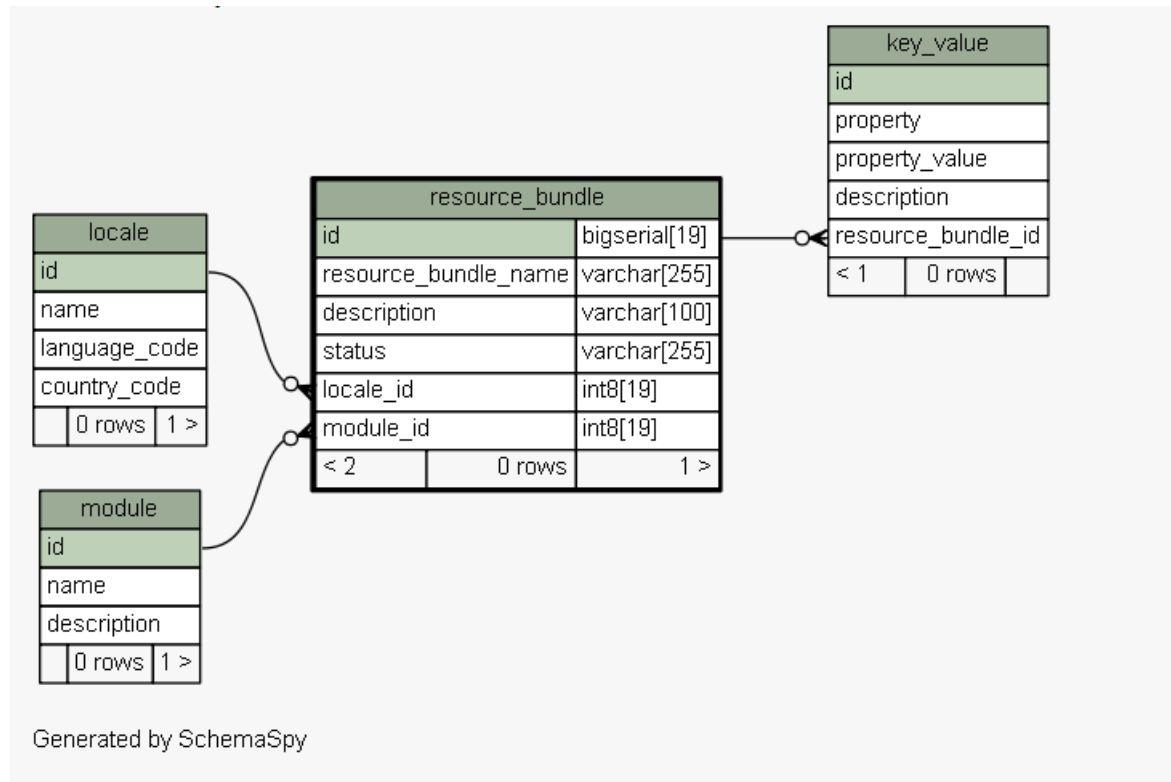
Note the locale table has a *country_code* field that won't be used in I18N app but is included for completeness to define a Locale. Don't worry about the bad naming conventions of tables and its primary keys. Those will be fixed latter.

Development Approach

The following is an overall picture of how I18N will be developed. Don't worry about the details as those will be covered latter on.

An empty JHipster app will be generated first as the template to develop everything.
A small database will be designed up front and modelled using *JHipster Domain Language(JDL)* file. Let's name this file *i18n.jd* from now on.

Then *jhipster:import-jdl* or *jhipster-uml* command will be run against *i18n.jd* within the project folder (i.e. jhipster_i18n) to generate all the frontend and backend code for the entities and its constraints declared in i18n.jd. By default, CRUD operations will be generated for each entity declared in i18n.jd. The generated code will include Liquibase XML files for creating database tables and its constraints.

Build & run the generated app to play around. Running the app will actually create the entire database tables & its constraints as specified in i18n.jd by using Liquibase master.xml file (which in turn uses many other XML files). Examine the tables and its constraints.

If not happy with the tables generated by Liquibase, refactor the Liquibase XML files generated above as required. Drop the database. Recreate it again then restart the app to generate all the tables and its constraints using the refactored Liquibase XML files. Table creation should be a onetime only process unless there is new change in master.xml file.

Since JDL can't be used to generate the following database changes at time of writing this, we will have to modify the generated JPAs from above to make it happen manually

- specify unique constraints on multiple columns of table

Finally implement all the frontend and backend logic required by function design above. Easy eh!

Generate an empty JHipster application

The first step will be to generate an empty JHipster app to be used as a template for doing all the development. See Create an application and all the related tutorials (e.g. Configure Intellij IDEA) on that page to get things up and running.

The figure below sums up the configurations I used to generate the I18N application. The important thing to note is we have to answer 'Yes' for the question 'Do you like to enable internationalization support?'. Choosing 'No' will defeat the purpose of implementing I18N app.

The figure also shows JHipster 3.4.2 was used. The app generation will take at least 3 to 5 minutes (at lease on my machine). Make sure to choose 'JWT' for authentication, Yes for Spring Websocket and EslaticSearch, Gradle for building the backend (if not familiar with Gradle). Why? This will give you an opportunity to get exposed to as much cool tech as possible.

Before this, I use Maven for building Java project. After implementing i18N with Gradle, I am afraid I have to say Gradle is better than Maven simply because it's simpler than Maven and requires less lines of code to achieve the same building tasks.

Figure 1: Generating a new JHipster app

⬧ Use '*git checkout -f commit-0*' to see the initial checkin of generated application.

Set up PostgreSQL database then run I18N app to populate database tables

Step 1: Install latest PostgreSQL and pgAdmin(admin tool for running SQL against PostgreSQL). I am using 9.4 and pgAdmin III

Step 2: Open 'pgAdmin' and run the following SQL to set up i18n database which will be used by the running I18N app.

> *create database i18n;*
> *create user i18n with password 'i18n';*
> *grant all privileges on database i18n to i18n;*

Note database, username and password all have the same value. I am following the great admin/admin username-password tradition.

Step 3: Configure I18N project datasource to use the database created above by opening *application-dev.yml* in IDEA, change the datasource configuration so it looks like

> …

datasource:

    url: jdbc:postgresql://localhost:5432/i18n
    name:
    username: i18n
    password: i18n

    …

Use *'git checkout -f commit-1'* to see the changes in application-dev.yml.

Step 4: Run I18N project and play around with it

The project can be run in console simply by cd into the project folder then run: *gradle bootRun.* If you only to do full, clean build and skip tests then run '*gradle clean build –x test*' in console.

Or if using IDEA, all you have to do is double-click the 'jhipster_18n [bootRun]' option in 'Gradle projects' View on the right. See figure below.
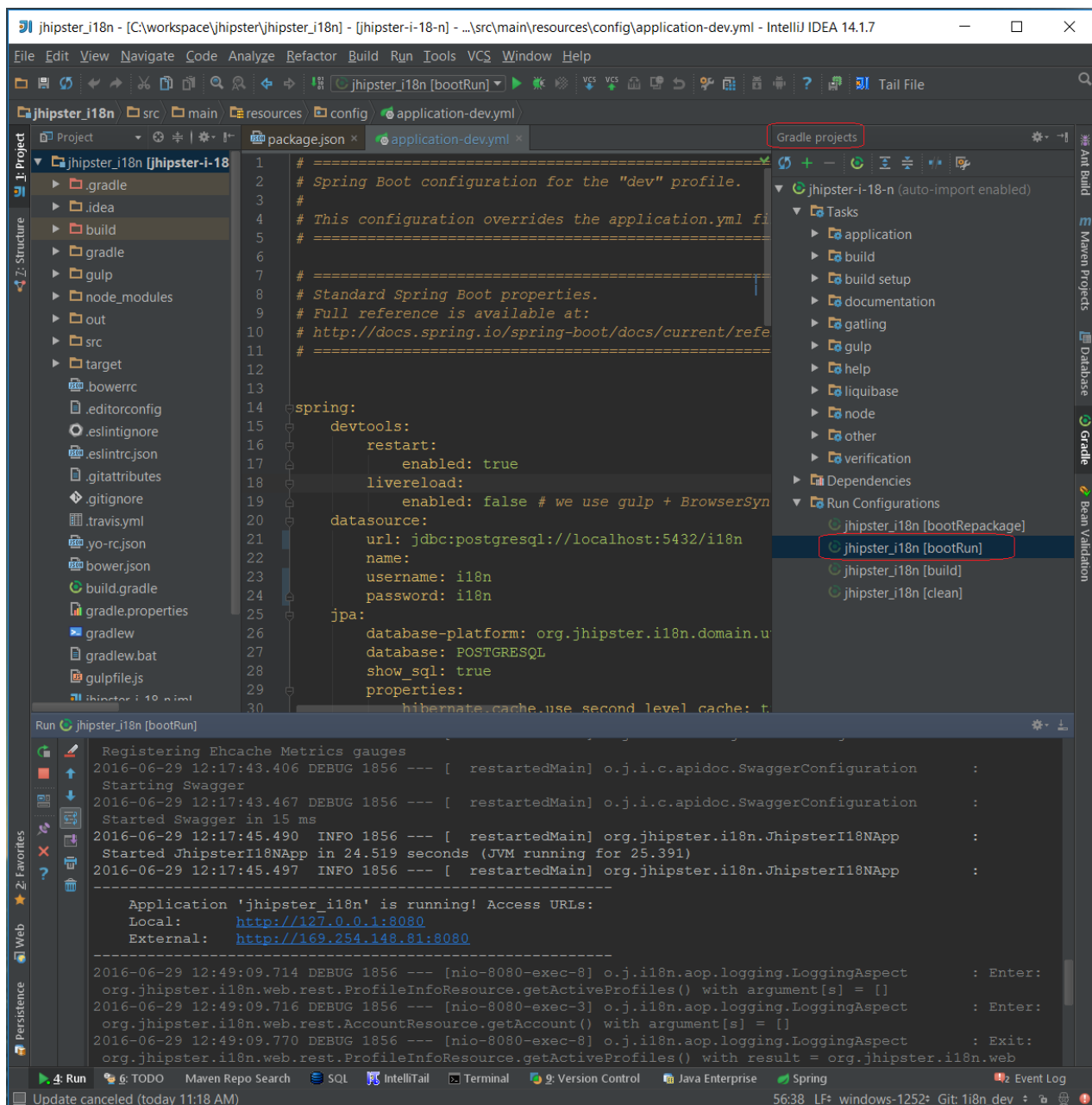
Figure 2: Running I18N in Intellij

Open a browser and visit *http://localhost:8080/#/* to see how awesome it is.

After the app is up & running, also run the command *'gulp'* in console so any changes in css/html/json files will be reflected automatically in browser. The default port used by gulp is 9000. Visit *http://localhost:9000/#/* in browser to see app running with Gulp.

Figure below shows the welcome page after logging in. Notice the highlighted 'Entities' menu is empty since we haven't implemented any entities yet.
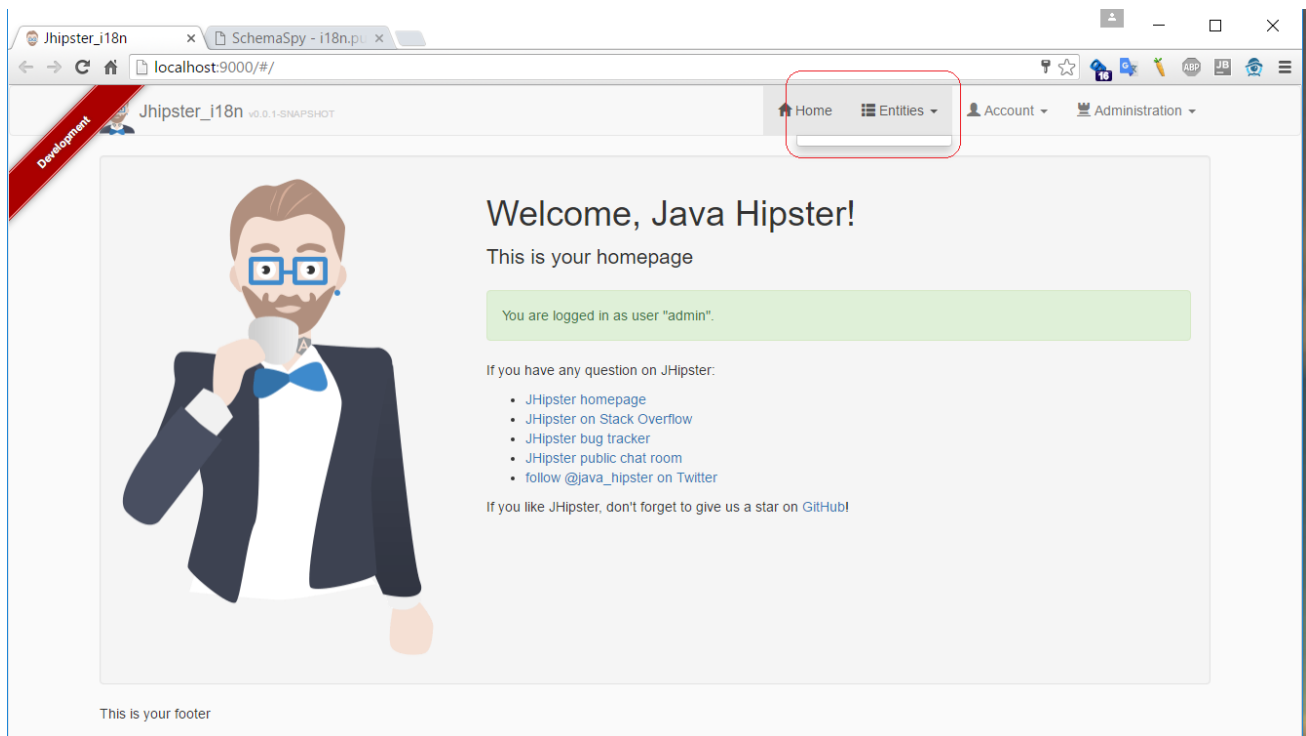
Figure 3: Welcome page after login

Without Gulp, any frontend changes you make won't be reflected automtomatically in browser. To see Gulp in action, make a change in html file then go back to browser to see the changes automatically. Yes, this is a must have feature during development.

Use SchemaSpy to examine the tables populated by I18N

Now go back to pgAdmin and use 'Object browser' window and see 7 tables generated by default after running *'gradle bootRun'*.
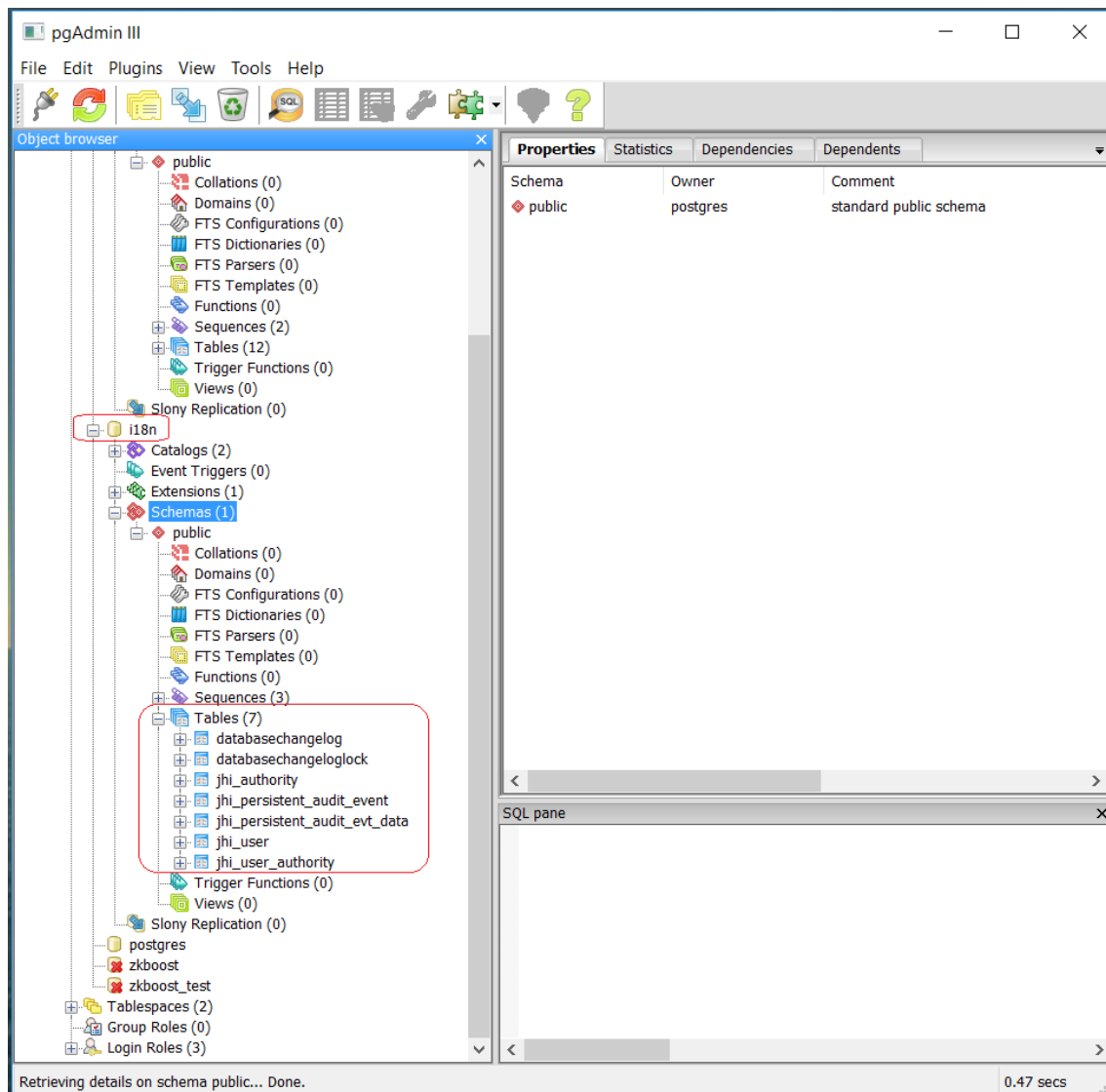
Figure 4: Default tables populated by running I18N app

Let's use SchemaSpy to generate the ER diagram of these table so we can have better understanding of their relationships. What's SchemaSpy? It's a Java-based open source tool used to generates the graphical representations of a database.

The easy way of using SchemaSpy is to use a GUI tool called SchemaSpyUI as in the following screenshots.

Figure 5.1: Specify JDBC driver, SchemaSpy jar file and JDBC settings

Figure 5.2: Specify where to output the generated HTML reports.
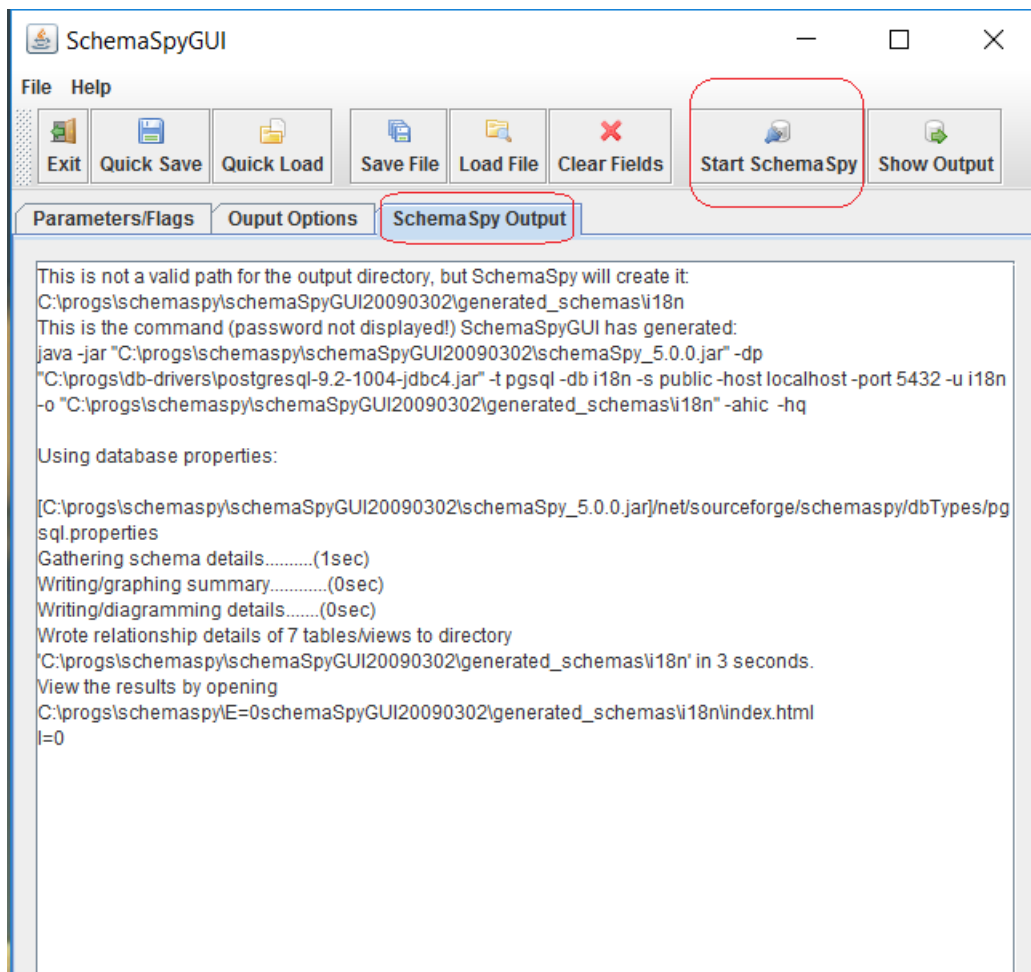
Figure 5.3: Click the big 'Start SchemaSpy' button to start generating reports. Watch the ouput to troubleshoot any error encountered.

The following diagrams generated by SchemaSpy show the default tables that come with any new JHipster app.

Figure 6.1: The default seven tables that come with any JHipster app
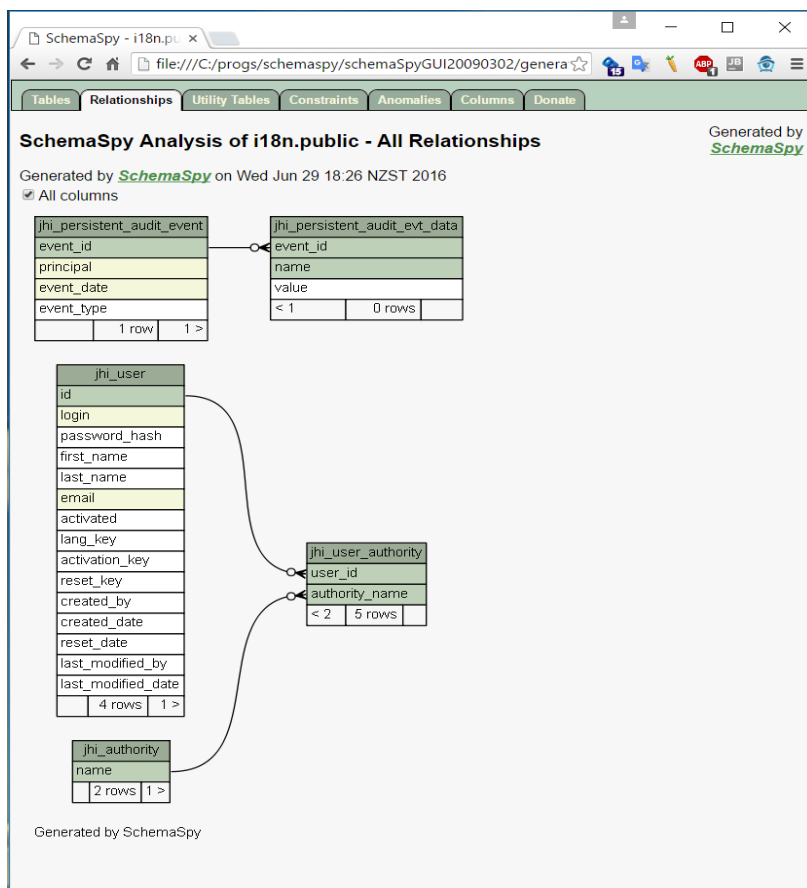
Figure 6.2: The relationships between these tables

SchemaSpy - Table i1 ×

file:///C:/progs/schemaspy/schemaSpyGUI20090302/generated_schemas/i18n/tabl

Tables | Relationships | Utility Tables | Constraints | Anomalies | Columns | Donate

Table i18n.public.jhi_user

Generated by
SchemaSpy

☐ Related columns ☐ Constraints ☐ Comments ☑ Legend

Column	Type	Size	Nulls	Auto	Default	Children	Parents
id	bigserial	19		√	nextval('jhi_user_id_seq'::regclass)	jhi_user_authority	
login	varchar	50					
password_hash	varchar	60	√		null		
first_name	varchar	50	√		null		
last_name	varchar	50	√		null		
email	varchar	100	√		null		
activated	bool	1					
lang_key	varchar	5	√		null		
activation_key	varchar	20	√		null		
reset_key	varchar	20	√		null		
created_by	varchar	50					
created_date	timestamp	29,6					
reset_date	timestamp	29,6	√		null		
last_modified_by	varchar	50	√		null		
last_modified_date	timestamp	29,6	√		null		

Legend:

SOURCEFORGE.NET

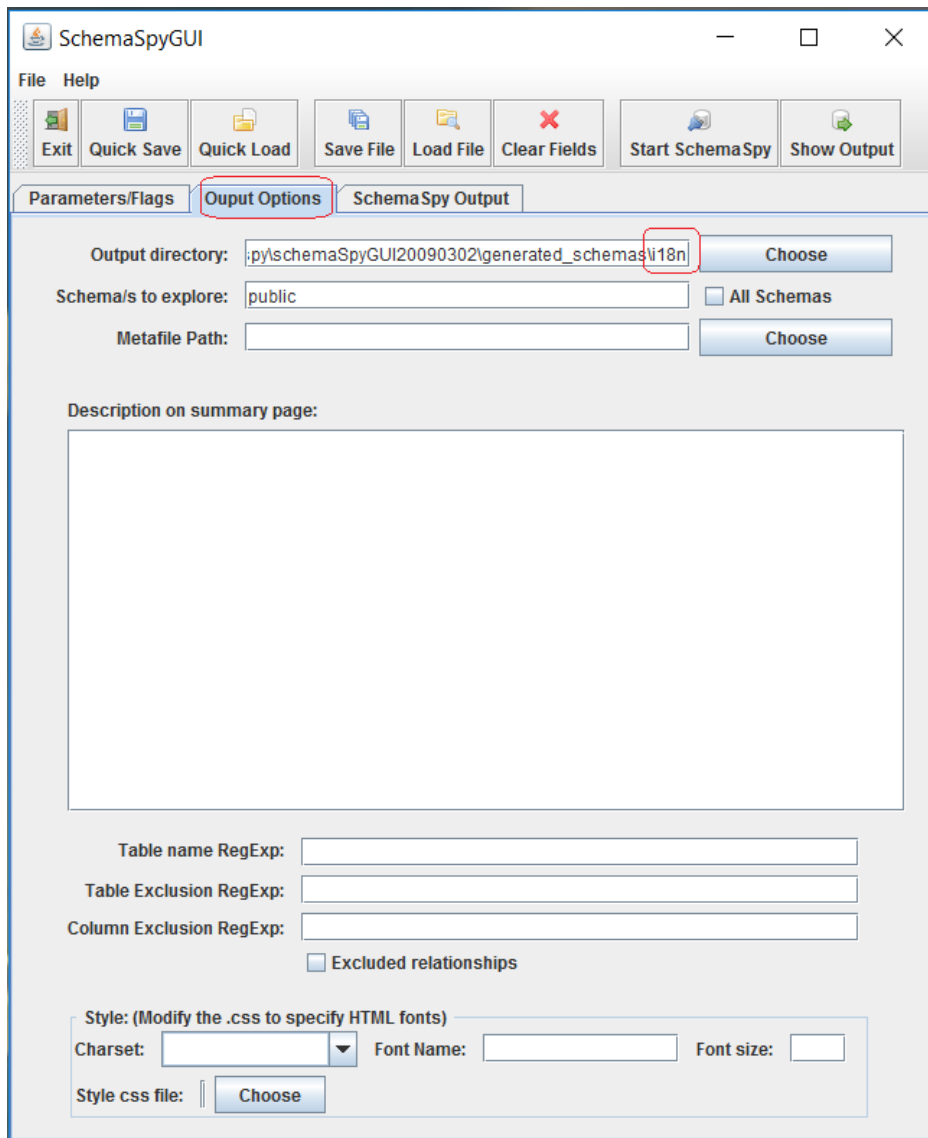| Primary key columns |
| Columns with indexes |
| Excluded column relationships |
| < n > number of related tables |

Table contained 4 rows at Wed Jun 29 18:26 NZST 2016

Indexes:

Column(s)	Type	Sort	Anomalies
id	Primary key	Asc	
email	Must be unique	Asc	This unique column is also nullable
login	Must be unique	Asc	
email	Must be unique	Asc	This unique column is also nullable
login	Must be unique	Asc	

Close relationships within ◉ one ○ two degrees of separation:

jhi_user	
id	bigserial[19]
login	varchar[50]
password_hash	varchar[60]
first_name	varchar[50]
last_name	varchar[50]
email	varchar[100]
activated	bool[1]
lang_key	varchar[5]
activation_key	varchar[20]
reset_key	varchar[20]
created_by	varchar[50]
created_date	timestamp[29,6]
reset_date	timestamp[29,6]
last_modified_by	varchar[50]
last_modified_date	timestamp[29,6]
< 0	4 rows	1 >

jhi_user_authority	
user_id	
authority_name	
< 2	5 rows

Generated by SchemaSpy

Figure 6.3: Closer look of jhi_user table's columns

As you can see, SchemaySpy is very useful and should be part of the power tools of any development team.

18

Model database design by using JHipster Domain Language(JDL)
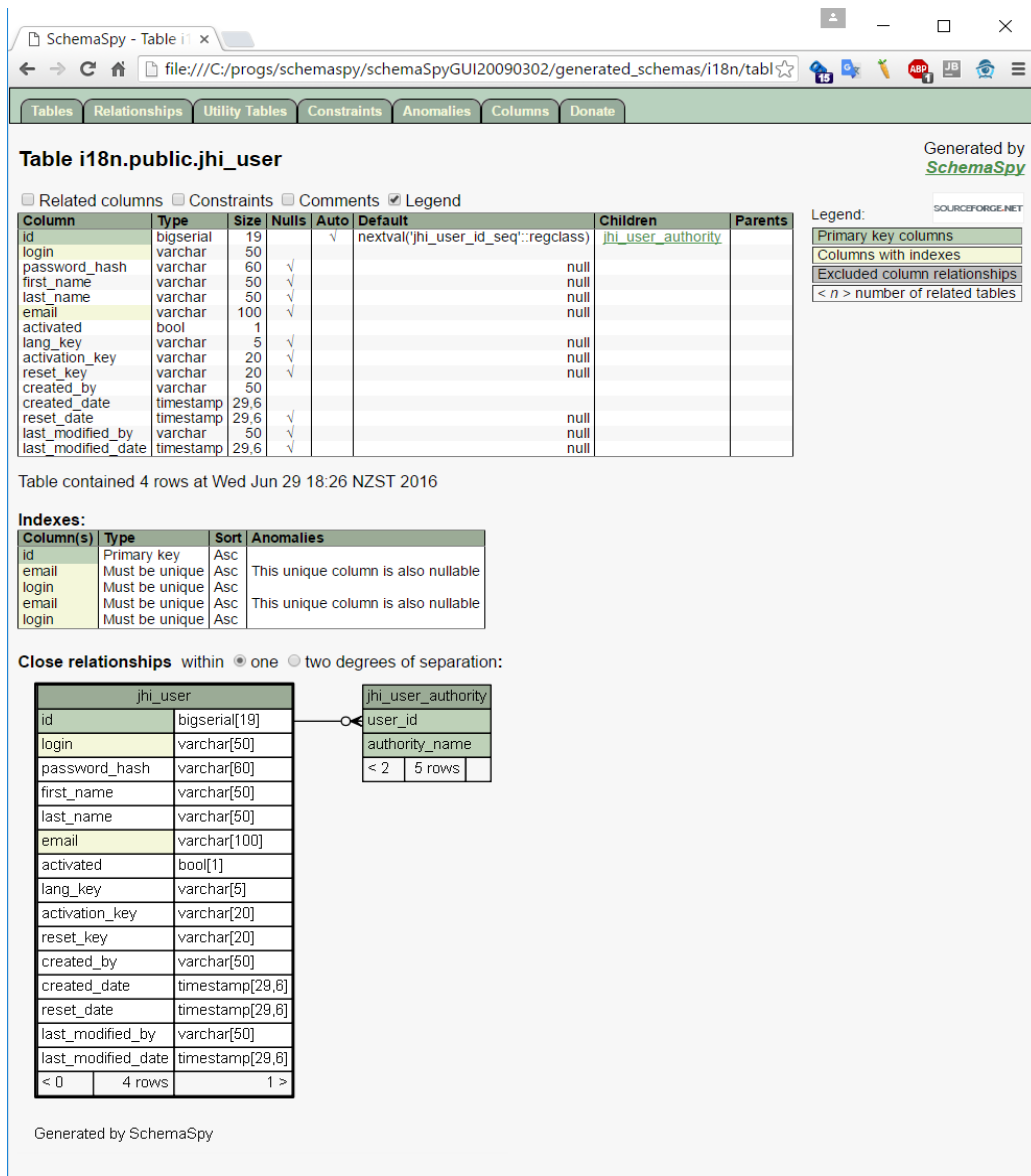
The next step is to create the JDL file *i18n.jd* to model the databaes design of I18N app so *jhipster-uml* or *jhipster:import-jdl* command can be used to generate all the frontend and backend code for the entities and its constraints declared in i18n.jd.

See JHipster Domain Language page and JDL Studio for how to write a JDL file. If one prefers to draw class diagrams using a UML editor so code can be generated from it, see JHipster-UML.

Note both jhipster:import-jdl and jhipster-uml commands can be used to generate code of jdl file but the main use of jhipster-uml is actually generating code off an XMI file (representing the drawn class diagrams) exported out from an UML editor.

After using both approaches, I would recommend using JDL as it's much simpler and a lot more things could go wrong with class diagram approach, e.g. the XMI file exported out from UML editor is not standard compliant or compatible with the parser in JHipster-UML.

The whole content of i18n.jd is listed below.

```
entity Locale {
  name String required,
  languageCode String required,
  countryCode String
}

entity Module {
  name String required,
  description String
}

enum ResourceBundleStatus {
    DISABLED, STATIC_JSON, DB_JSON
}
entity ResourceBundle {
 resourceBundleName String,
 description String maxlength(100)
 status ResourceBundleStatus
}

entity KeyValue {
  property String required,
  propertyValue String required,
  description String
}

relationship ManyToOne {
  ResourceBundle{Locale} to Locale
}
relationship ManyToOne {
  ResourceBundle{Module} to Module
}

relationship ManyToOne {
  KeyValue{ResourceBundle} to ResourceBundle
}
```

Object-Relational mapping of i18n.jd explained

I will explain more about the object-relational (OR) mapping in i18n.jd here. First of all, read JHipster Domain Language so I don't have to explain.

Now what may seem questionable by developers familiar with OR mapping in Hibernate is ResourceBundle entity doesn't declare a *One-to-Many* relationship with KeyValue entity. Instead, it's the KeyValue that declares a *Many-to-One* relationship to ResourceBundle.

After all, Hibernate allows the parent entity (i.e. ResourceBundle) of One-to-Many relationship to do cascade delete/save-update operation to the child records(the many side in KeyValue entity). Also, a One-to-Many mapping would allow a retrieved ResourceBundle record to contain all KeyValues records automatically in a collection. No need for a second call to database to get all KeyValue records attached to the ResourceBundle. In case reader is not familiar with Hiberante cascade operations, see here, here and here.

Well, for a web app that doesn't expose data through restful services, using One-to-Many mapping may work fine. But as I have found out the hard way, it's more trouble than its worth here. Let me explain.

By default, a One-to-Many JDL mapping will generate a bidirectional relationship so both ResourceBundle and KeyValue can access each other in JPA. That is a little bad for performance to begin with.

```
Entity ResourceBundle { … }
entity KeyValue { … }
relationship OneToMany {
  ResourceBundle{ KeyValue } to KeyValue{ ResourceBundle }
}
```

A One-to-Many mapping between ResourceBundle to KeyValue in JDL

Now when you modify the resource bundle page(i.e. resource-bundle-dialog.html) to also display key value records attached to it, you will see the infamous *org.hibernate.**LazyInitializationException***. It happens as frontend tries to load KeyValue records attached to ResourceBundle but the database connection used to retrieve ResourceBundle is already closed.

Don't worry. It can be fixed easily by eager fetching all KeyValue records whenever the ResourceBundle is retrieved as shown below in ResourceBundle JPA. Note the fix is highlighted bold.

> **@OneToMany**(**mappedBy="resourceBundle"**, cascade=CascadeType.ALL, **fetch=FetchType.EAGER**, orphanRemoval=true)
> private List<KeyValue> keyValues = new ArrayList<KeyValue>();

This does solve the problem but introduces a much worse problem, the infinite recursion of loading data between ResourceBundle and KeyValue (as they reference each other in JPA) in restful services. I tried most of the suggested workarounds (e.g. *@JsonIgnore*) as in here, here and here. All the fix comes with its own side effect or introduces new problem. In the end, I come to the realization that I could avoid all these pain by simply using Many(i.e. KeyValue) to One(i.e. ResourceBundle) mapping in the 1st place.

Another way of fixing the problem without *eager-fetching* is to introduce a DTO layer and uses DTOs to display data in frontend instead of JPA. The DTO layer (e.g. ResourceBundleDTOService.java) will then be responsible for loading KeyValues, convert it to KeyValueDTOs then put it on ResourceBundleDTO). However,

this is overkill in demo so no go there. See [DTO](#) for more concise definition. See [here](#) for using DTO in JHipster.

The catch of using Many-to-One mapping without DTO is when a ResourceBundle page needs to display a list of its KeyValues in a table, a second restful service request will be required to get all KeyValue data. But that's way simpler than going through the above problems. The latter tutorial will show how to do this exactly.

So there you have it. The lesson is think hard before you decide to use One-To-Many OR mapping in restful service. Often a Many-to-One mapping is much simpler to use and maintain than a One-to-Many mapping in my experience.

Code generation with JDL

First off, install JDL locally in current project by running the following command in jhipster_i18n project folder:

npm install jhipster-domain-language --save

The package.json file will be updated to include *jhipster-domain-language* dependency after install.

Then run '*yo jhipster:import-jdl i18n.jh*' and the code should be generated in Theory. I said in theory because at the time of writing this (2-July-2016), this command is not working in JHipster 3.4.2. Luckily, we can use '*jhipster-uml i18n.jh*' as in an interim workaround.

As far as I can tell, the code generated from *jhipster-uml* is pretty much the same as *jhipster:import-jdl* and everything still works as expected when running i18n app.

Ues '*git checkout –f commit-3*' to see both the frontend & backend code generated

Examine the new database tables generated from JDL

To generate the new tables and its constraints specified in i18n.jd, drop the database 1st, then recreate the database with

> *create database i18n;*
> *grant all privileges on database i18n to i18n;*

Then start up I18N app again to populate all the tables automatically. If everything goes, CRUD operations for each entity declared in i18n.jh will be functioal & working.

So what's happening behind the scene? Running '*yo jhipster:import-jdl i18n.jh*' resulted in updating Liquibase **master.xml** to reference all the newly generated XML configurations that would create new tables and its constraints.

Ues '*git checkout –f commit-2*' to see the Liquibase XML files changes mentioned above

To examine the new tables and its relationships, let's use SchemaSpy again to gernate the ER diagrams.

SchemaSpy - Table i1 ×

file:///C:/progs/schemaspy/schemaSpyGUI20090302/generated_schema

Tables | Relationships | Utility Tables | Constraints | Anomalies | Columns | Donate

Table i18n.public.resource_bundle

Generated by
SchemaSpy

☐ Implied relationships ☑ Related columns ☑ Constraints ☐ Comments ☑ Legend

Column	Type	Size	Nulls	Auto	Default	Children
id	bigserial	19		√	nextval('resource_bundle_id_seq'::regclass)	key_value.resource_bundle_id fk_keyval
resource_bundle_name	varchar	255	√		null	
description	varchar	100	√		null	
status	varchar	255	√		null	
locale_id	int8	19	√		null	
module_id	int8	19	√		null	

Table contained 0 rows at Thu Jun 30 12:58 NZST 2016

Indexes:

Column(s)	Type	Sort	Constraint Name
id	Primary key	Asc	pk_resource_bundle

Close relationships:

Generated by SchemaSpy

Figure 7: New tables and its relationships created by Liquibase after starting I18N app

One may notice the names of all the tables are un-capitalized. Even the name of all primary keys is simply 'id'. It's enough to make a DB admin cringe. We will fix all these soon.

23

Refactor the new tables generated by JDL

To fix uncapitalized table names and bad primary key naming above, we need to change the Liquibase XML files used in master.xml. Let's take the XML file (i.e. 20160629121218_added_entity_Locale.xml) that's used to create Locael table as an example.

Before change

```
    ….
    <changeSet id="20160629121218-1" author="jhipster">
        <createTable tableName="locale">
          <column name="id" type="bigint" autoIncrement="${autoIncrement}">
            <constraints primaryKey="true" nullable="false"/>
          </column>
           ….
          </createTable>
      </changeSet>
</databaseChangeLog>
```

After change

```
    ….
    <changeSet id="20160629121218-1" author="jhipster">
        <createTable tableName="Locale ">
          <column name="locale_id " type="bigint" autoIncrement="${autoIncrement}">
            <constraints primaryKey="true" nullable="false"/>
          </column>
          …
          </createTable>
      </changeSet>
</databaseChangeLog>
```

The only changes are the red font highlighted bold.The other tables will have the same treatment. Easy!

For the changes to take effect, stop i18n app, drop i18n database, recreate it again using SQL from '**Set up PostgreSQL database then run I18N app to populate database tables'.** Then start I18N app again.

To convince ourselves the database refactoring is a success, let's generate ER diagrams using SchemaSpy again. See figure below.

Figure 8: Refactored database after changing Liquibase XML files

Exactly what we want. Now there is good news and bad news. Good news – database refactoring successful. Bad news – all the front end and back end code would be broken as result of database refactoring. That's OK. We will fix it soon.

It's much better to sort out database issue ASAP. The later we fix it, the more impact and pain it will cause in codebase.

'*git checkout –f commit-6*' contains Liquibase XML files changes for database refactoring

Since all the database refactoring is done through Liquibase, just to mention JHipster has barely scratched the surface of Liquidbase which basically uses XML as **database independent platform** to manage, track and version database changes. This measn the same XML files can work for all databases supported by Liquibase!

So just to list some Liquibase links before moving on:

- To use liquibase-gradle-plugin in IntelliJ
- Liquibase workshop – gives some hands-on experience
- Gentle introductions to Liquibase
- Database-independent script in Liquibase
- Liquibase change set that runs every time

And finally visit their official website for more info at http://www.liquibase.org/.

Fixing broken code resulted from refactoring database

If we start up I18N app again, it will seem to run without problem. But as soon as we try to save something, say a Locale record, boom, error message.
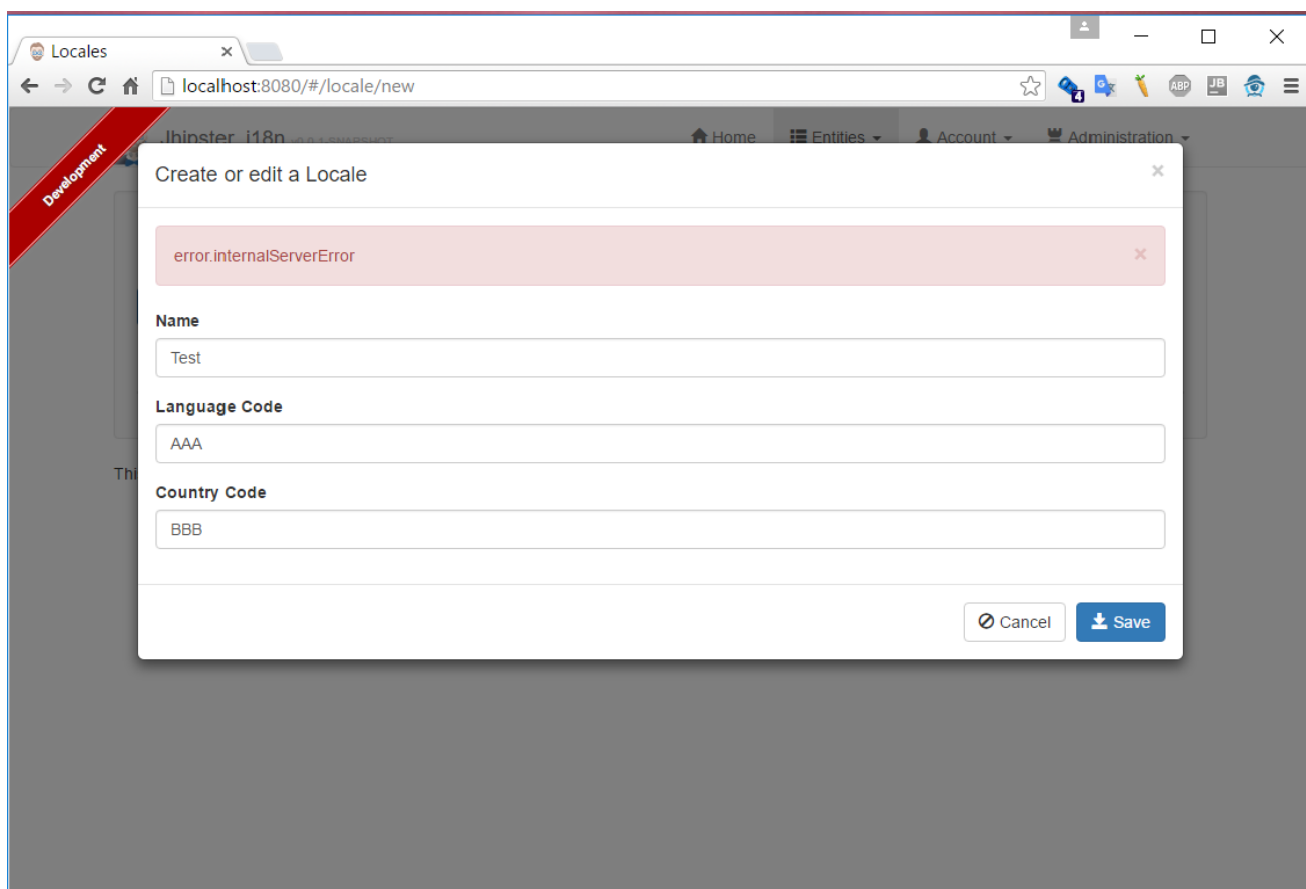

Figure 9: Error message when trying to save a Locale record.

Checking console will see the exception

…
Caused by: org.postgresql.util.PSQLException: ERROR: relation "locale" does not exist
 Position: 141
        at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2284)
        at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2003)
        at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:200)
        at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:424)
        at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:161)
        at org.postgresql.jdbc.PgPreparedStatement.executeQuery(PgPreparedStatement.java:114)
        at com.zaxxer.hikari.pool.ProxyPreparedStatement.executeQuery(ProxyPreparedStatement.java:52)
        at com.zaxxer.hikari.pool.HikariProxyPreparedStatement.executeQuery(HikariProxyPreparedStatement.java)
        at org.hibernate.engine.jdbc.internal.ResultSetReturnImpl.extract(ResultSetReturnImpl.java:82)
        ... 161 common frames omitted

 ….

Good. It's failing as expected. If not, be afraid. We may still be within a dream.

The fix will need to be done at both backend (e.g. JPA Locale.java) and frontend if any. I will only use Locale entity as an example here. Reader can check out the rest of the fix in Git commit latter.

After burning some hours at trying to make JPA recognize the capitalized table names, I decided to leave that for now and revert back the Liquibase changes for capitalizing table names.

Why? The problem turns out to be too exciting as below.

Turns out by default, PostgreSQL 9.4 uses lowercase to access table name instead of uppercase. This means running SQL 'select * from Locale;' in pgAdmin will result in error message. One has to use double-quote around the table as 'select * from "Locale";' to make it work.

Implications? In Locale.java, changing @Table(name = "locale") to @Table(name = "Locale") simply won't work.

Googling leads to a StackOverflow post suggesting @Table(name = "\"Locale\""). Tried that, doesn't work. Then another StackOverflow thread suggests using

```
        spring.jpa.hibernate.naming_strategy=org.hibernate.cfg.EJB3NamingStrategy
```

In JHipster, this will be set in application-dev.yml. Still doesn't work. Then I saw the class FixedPostgreSQL82Dialect is extending PostgreSQL82Dialect but I am using PostgreSQL 9.4 so naturally I want to use the latest which is PostgreSQL94Dialect. So I ended up updating hibernate-core to the latest. Doing build seems OK but running the app spilled out new error messages and opened a can of worms… Remind me to use MySQL next time.

Databae refactoring for renaming primary keys still stands so we still need to fix the backend code in JPA. One unexpected bonus is that no frontend code needs to be changed.

After the fix, drop database then re-create it again as before. Start up I18N app again to populate the database. Testing shows everything is working fine now.

See '*git checkout –f commit-7*' for the fix discussed above.

Setting unique constraints in JPAs

At this stage, all the data(i.e. ResourceBundle/KeyValue/Module/Locale) can be inserted, updated or deleted successfully from a running I18N app. Before jumping to work on the frontend, let's fix the unique constraint issue in ResourceBundle entity.

Remember back in Functional Design, each ResourceBundle record needs to have a unique combination of language *code* and *name* properties. If attempting to insert a ResourceBundle record having the same language *code* and *name* as an existing one, backend should throw exception so frontend could display a proper error message. Easyyy. Or is it?

First stop, creating a ResourceBundle record. When trying to, we see the problem of Locale and Module Drop down lists displaying a number (the primary key) instead of anything meaningful. Let's fix that first.
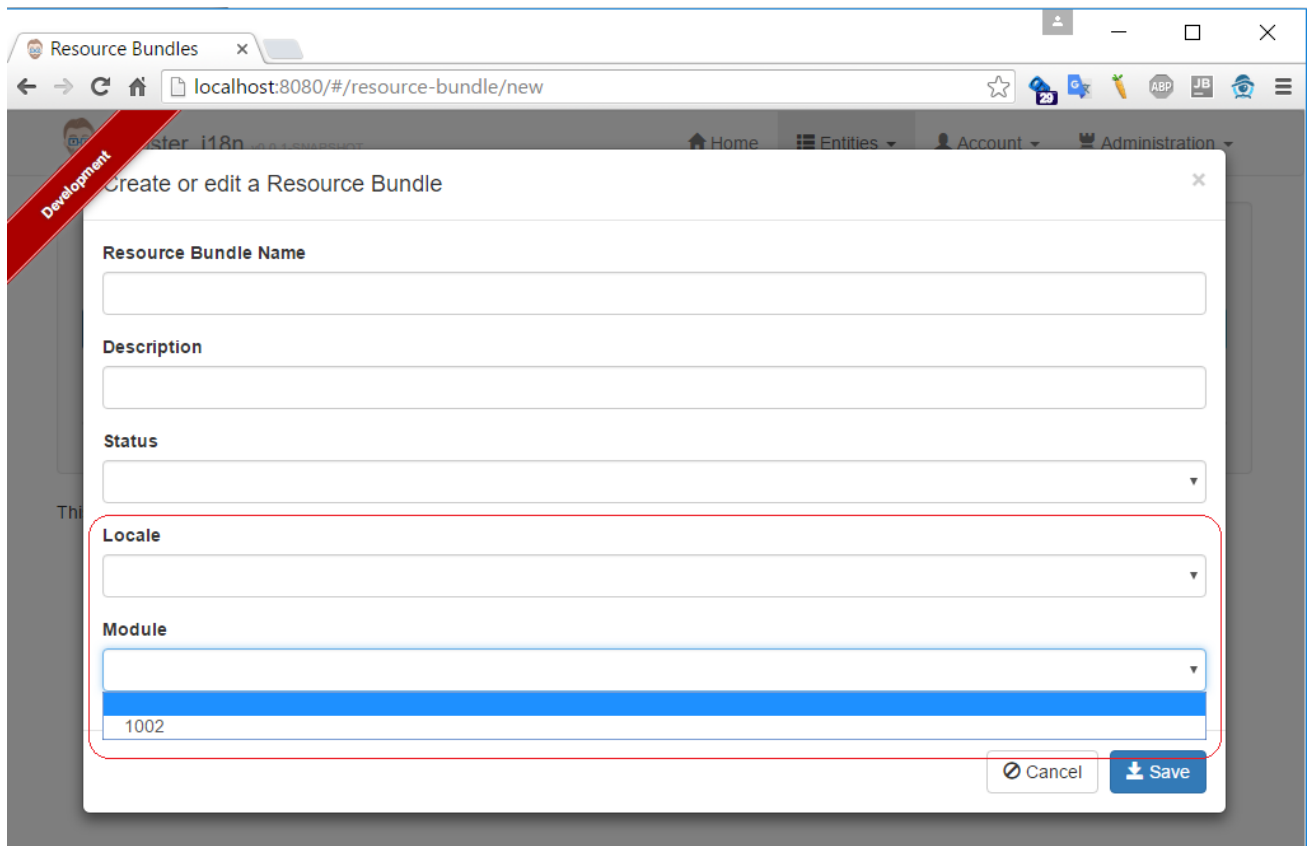
Figure 10: Locale and Module drop down lists displaying primary key.

To fix it, open *resource-bundle-dialog.html* and look for the HTML code that displays Locale & Module drop down lists. The culprit is highlighted bold below

```
..
<div class="form-group">
    <label translate="jhipsterI18NApp.resourceBundle.locale" for="field_locale">Locale</label>
    <select class="form-control" id="field_locale" name="locale" ng-model="vm.resourceBundle.locale"
        ng-options="locale as locale.id for locale in vm.locales track by locale.id">
      <option value=""></option>
    </select>
</div>
<div class="form-group">
    <label translate="jhipsterI18NApp.resourceBundle.module" for="field_module">Module</label>
    <select class="form-control" id="field_module" name="module" ng-model="vm.resourceBundle.module"
        ng-options="module as module.id for module in vm.modules track by module.id">
      <option value=""></option>
    </select>
</div>
...
```

The fix is simply changing *locale.id* to *locale.name* and change *module.id* to *module.name.*
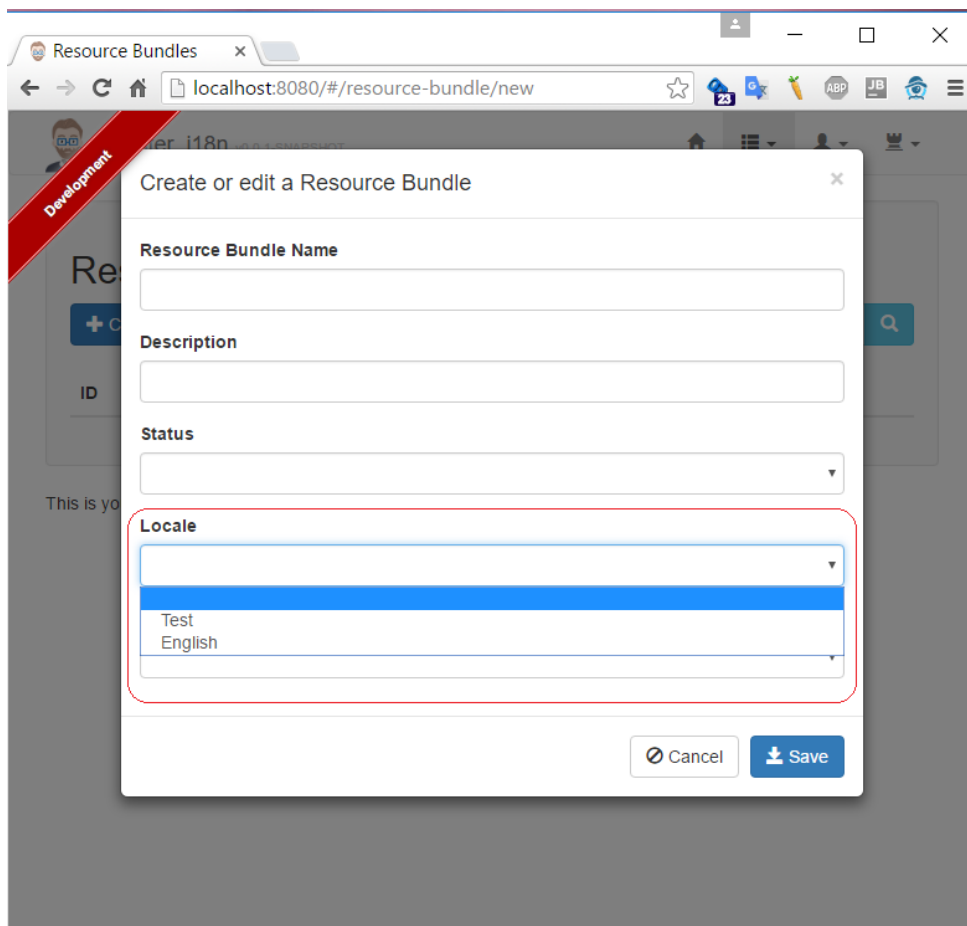
Figure 10: After fix. Works as expected now.

Now we implement the actual unique constraints on ResourceBundle. The way to do that in ResourceBundle JPA is to use **@UniqueConstraint.** So we added that in JPA as highlighted red below

```
@Entity
@Table(name = "resource_bundle", uniqueConstraints = @UniqueConstraint(columnNames = {"locale_id",
"module_id" }))
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
@Document(indexName = "resourcebundle")
public class ResourceBundle implements Serializable {
        ...

    @ManyToOne( fetch=FetchType.EAGER  )
    @JoinColumn(name="locale_id", nullable=false)
    private Locale locale;

    @ManyToOne( fetch=FetchType.EAGER )
    @JoinColumn(name="module_id", nullable=false)
    private Module module;
        ...
```

Cool! Now restart the server and try to create two ResourceBundles having the same Locale and Module, see what happens. No error messages in console or browser. Something is wrong.

After some googling, we should arrive at this [StackOverflow link](#) which makes it clear that

> …
>
> The only effect of specifying @UniqueConstraint is that a unique constraint will be added to the database schema if you are using your JPA provider's schema generation functionality. If not then it has absolutely no effect.
>
> …

So double-click the Liquibase task '*liquibaseDiffChangelog*' in 'Gradle projects' panel on the right hand side of Intellij (or type '*gradle liquibaseDiffChangelog*' in console) to generate the Liquibase XML change log containing the generated unique constraints specified by JPA change above. Then update master.xml to include the generated XML change log. Then stop server, drop i18n database, recreate it again as before. Restart server and everything will be fine and dandy.

Don't believe me? Checking resource_bundle table's constraints in pgAdmin will show the new constraint. Let's repeat the same test as before in browser. This time we get an error message in browser. Checking console also shows the **DataIntegrityViolationException.** O-Yeah!

> …
> org.springframework.dao.**DataIntegrityViolationException**: could not execute statement; SQL [n/a]; constraint [resource_bundle_locale_id_module_id_key]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement
> at
> org.springframework.orm.jpa.vendor.HibernateJpaDialect.convertHibernateAccessException(HibernateJpaDialect.java:259)
>   at
> org.springframework.orm.jpa.vendor.HibernateJpaDialect.translateExceptionIfPossible(HibernateJpaDialect.java:225)
>   ...
> Caused by: org.postgresql.util.PSQLException: ERROR: **duplicate key value violates unique constraint**
> "resource_bundle_locale_id_module_id_key"
>   **Detail: Key (locale_id, module_id)=(1000, 1001) already exists.**
>   ....
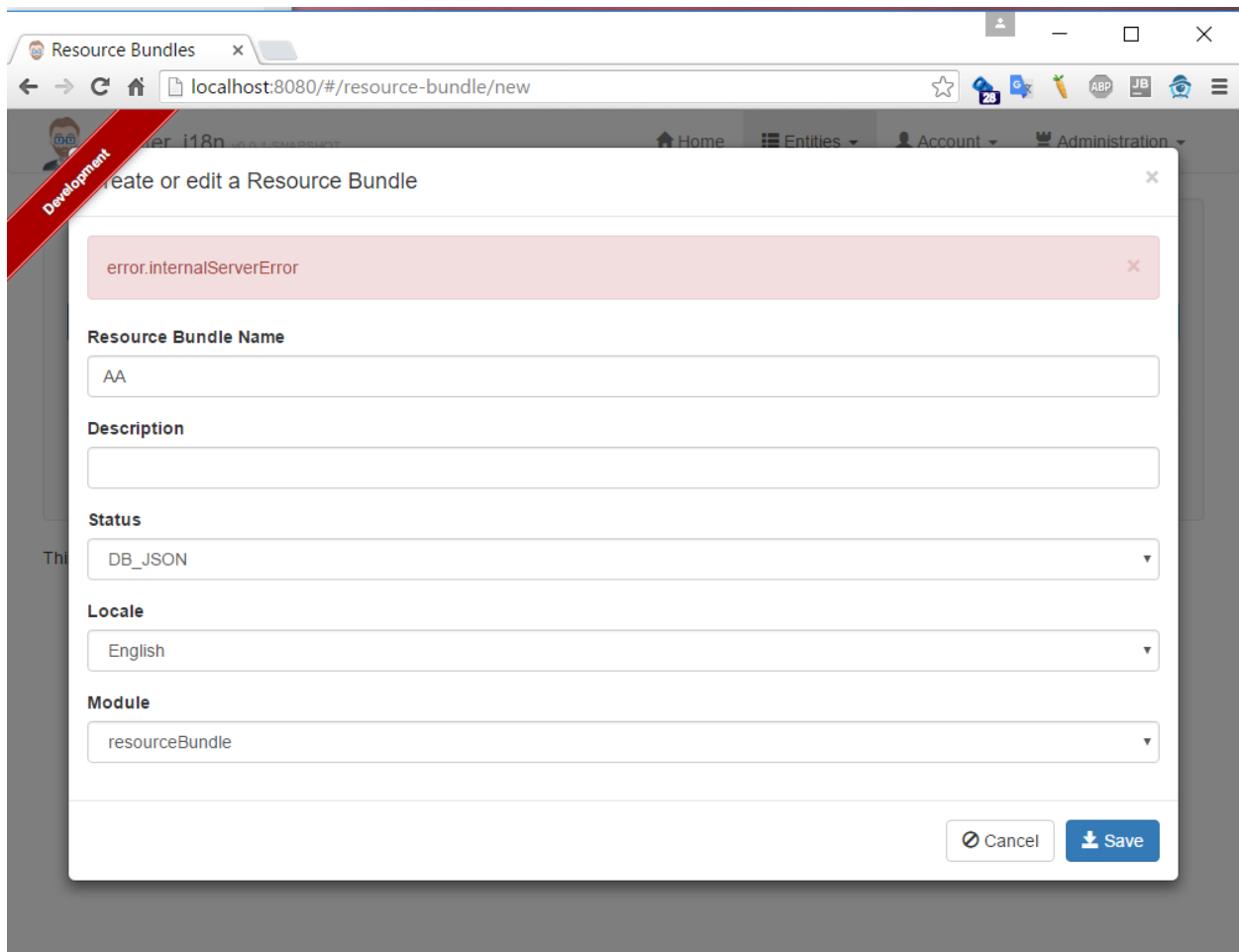
Figure 11: Get error message when tring to create a Resource Bundle having the same Locale and Module as an existing one, i.e. breaks unique constraints of Locale & Module

Yes, it's working as expected, more or less. The error message in browser is not useful at all. A more meaningful error message has to be displayed. Imagine a data constraint error happens in production, client has no idea why it happens, jumps up and down and give you this screenshot. You will also jump up and down if you can't decipher(guess is more realistic) anything from their server log files.

Also by default, the red alert box in Figure 11 will disapear after five sec, this is hardly useful. The next section will attempt to fix all these.

'*git checkout –f commit-8*' contains fix in this section.

Display meaningful error message in alert box (see Figure 11 above for clarification)

The heading of this section may sound easy. The actual roadblocks are not. Let's break it down.

1. The alert box needs to stay after 5 sec till user closes it manually through X icon
2. The alert box needs to display meaningful error message for debugging
3. *The Java backend needs to catch **DataIntegrityViolationException** caused by violation of ResourceBundle's unique constraints (or data constraints of any table) and convert that exception to some meaningful error message for alert box in the frontend to display*
4. *And anything that can go wrong by Murphy's Law…*

Too easy! Just shut down everything.

To fix the **first problem**, let's start with resource-buundle-dialog.html(the page showing the problem), look really hard and see

```
<jhi-alert-error></jhi-alert-error>
```

This is what's called an [Angular Component](#) that is a special kind of **Angular Directive**(memory flash back ~ [here](#), [here](#) … [here](#)).

The *<jhi-alert-error>* is:
- Using the HTML template defined in **alert-error.directive.js** to display the alert box.
- Using the [Angular Service](#) ([link](#)) *AlertService* within its controller to generate actual alert messages for display in alert box. The AlertService used is actually defined in **alert.service.js**.

To be really a pro, we also look at *Notification System* section of this [JHipster page](#) to see how AlertService is used and check out [ui-bootstrap alerts](#), which is used to implement the alert box throught the geneated JHipster app.

Now we are ready to fix the **first problem**. Open up *alert-error.directive.js*, dive down and locate the default timeout set at the *addErrorAlert* function

```
function addErrorAlert (message, key, data) {
    key = key && key !== null ? key : message;
    vm.alerts.push(
        AlertService.add(
            {
                type: 'danger',
                msg: key,
                params: data,
                timeout: 5000,
                toast: AlertService.isToast(),
                scoped: true
            },
            …
```

Replace that timeout line with

```
timeout = 0; // no timeout
```

Also repeat the same fix in alert.service.js for info messages displayed in alert box

```
…
function getService ($timeout, $sce,$translate) {
    var toast = this.toast,
        alertId = 0, // unique id for each alert. Starts from 0.
        alerts = [],
        // timeout = 5000; // default timeout
        timeout = 0 ; // No timeout

…
```

Repeat the test from last section again in browser. The alert box stays forever now.

The second and third problems are strongly tied so we basically need to catch **DataIntegrityViolationException** and extract some root cause error message if any. If no error message from the caught **DataIntegrityViolationException,** then display a default error message.

In JHipster world, almost all the exceptions are caught globally in *ExceptionTranslator* class annotated with @ControllerAdvice. Any method that's annotated with *@ExceptionHandler* within *ExceptionTranslator* will catch a specific exception and decides how to respond to the request.

More memory flashback Spring Boot Error Responses and REST API error return good practices.

So we will simply add a new method that will catch all **DataIntegrityViolationException,** extract root cause message from it, put that error message in ErrorDTO, return ErrorDTO back to client with HTTP Error Status 409 Conflict. Job done!

```
@ExceptionHandler(DataIntegrityViolationException.class)
@ResponseBody
@ResponseStatus(HttpStatus.CONFLICT)
public ErrorDTO processDataIntegrityViolationException(DataIntegrityViolationException exception) {

    String errorMsgPrefix = "Data Integrity Violation error occurred: ";
    log.error(errorMsgPrefix + " {}", exception.getMessage(),  exception);

    String errorMessage = null;
    if (exception.getRootCause() != null) {
        errorMessage = exception.getRootCause().getMessage();
    } else if (exception.getMessage() != null) {
        errorMessage = exception.getMessage();
    } else {
        errorMessage = exception.toString();   // set default error message
    }

    errorMessage = errorMsgPrefix + errorMessage;
    ErrorDTO errorDTO = new ErrorDTO(errorMessage, exception.getMessage());
    return errorDTO;
}
```
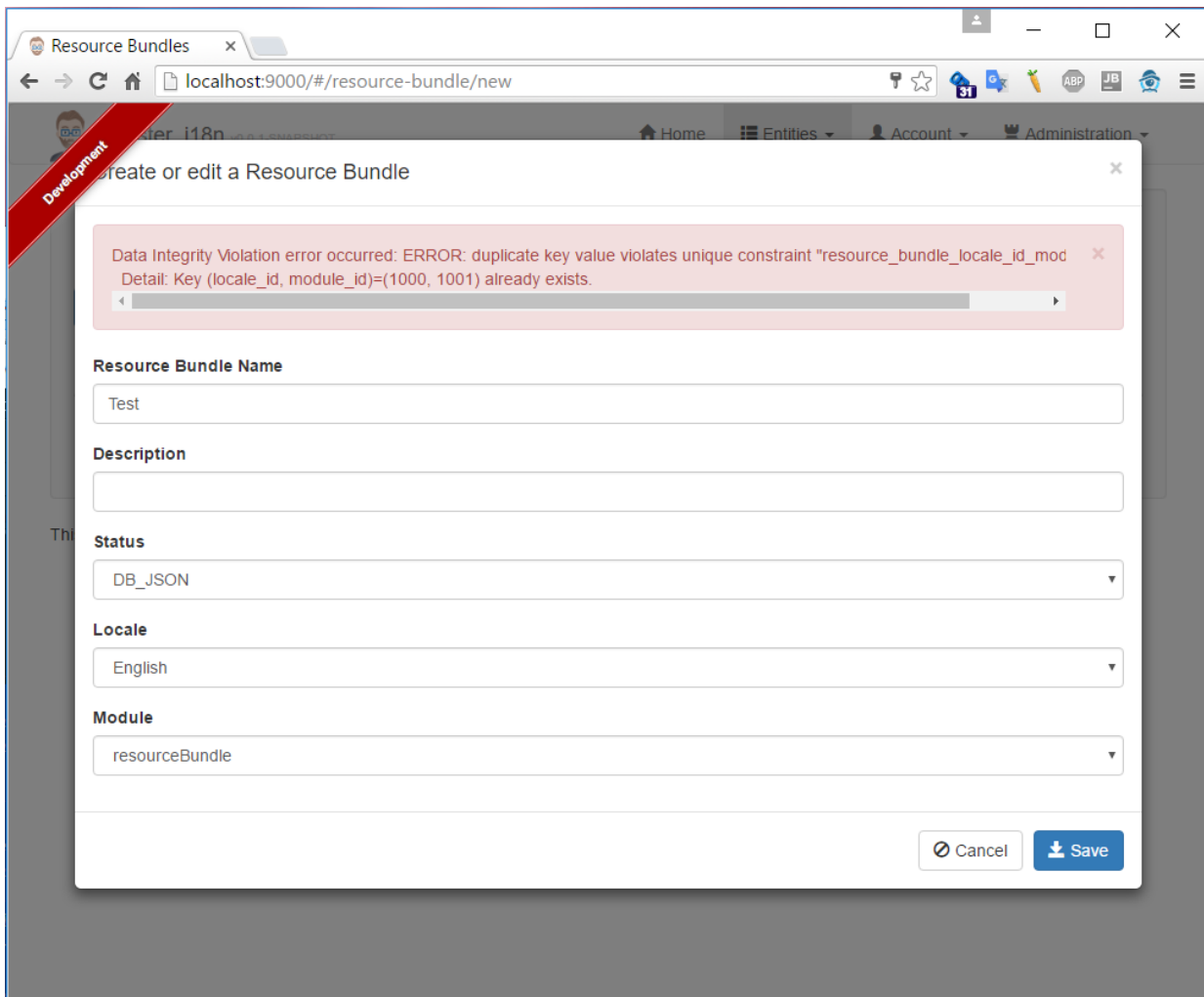
Figure 12: Error message from violating unique constraint of Locale and Module

Yes, this is sensible error message now. May not be user friendly but enough to show root cause of error. Also the exception will appear in the log file now.

Speaking of Log file

By default in JHipster 3.4.2, there is no log file. All the log will be output to console. To change this, open logback-spring.xml and uncomment the following block. Modify as required to fit your own needs.

```
<!--
  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>logFile.%d{yyyy-MM-dd}.log</fileNamePattern>
      <maxHistory>90</maxHistory>
    </rollingPolicy>
  </appender>
  .....
-->
```

See '*git checkout –f commit-9*' for fix in this section.

Polish up form validations

The default look and feel of frontend and form validations generated by JHipster using AngularJS and Bootstrap is good but could still use some clean-up so lets do it now.

Let's pick Locale page as an example as seen in Figure 13. All the mandatory fields have gray text underneath it to indicate it's required and the 'Save' button will be disabled till the mandatory fields are filled.
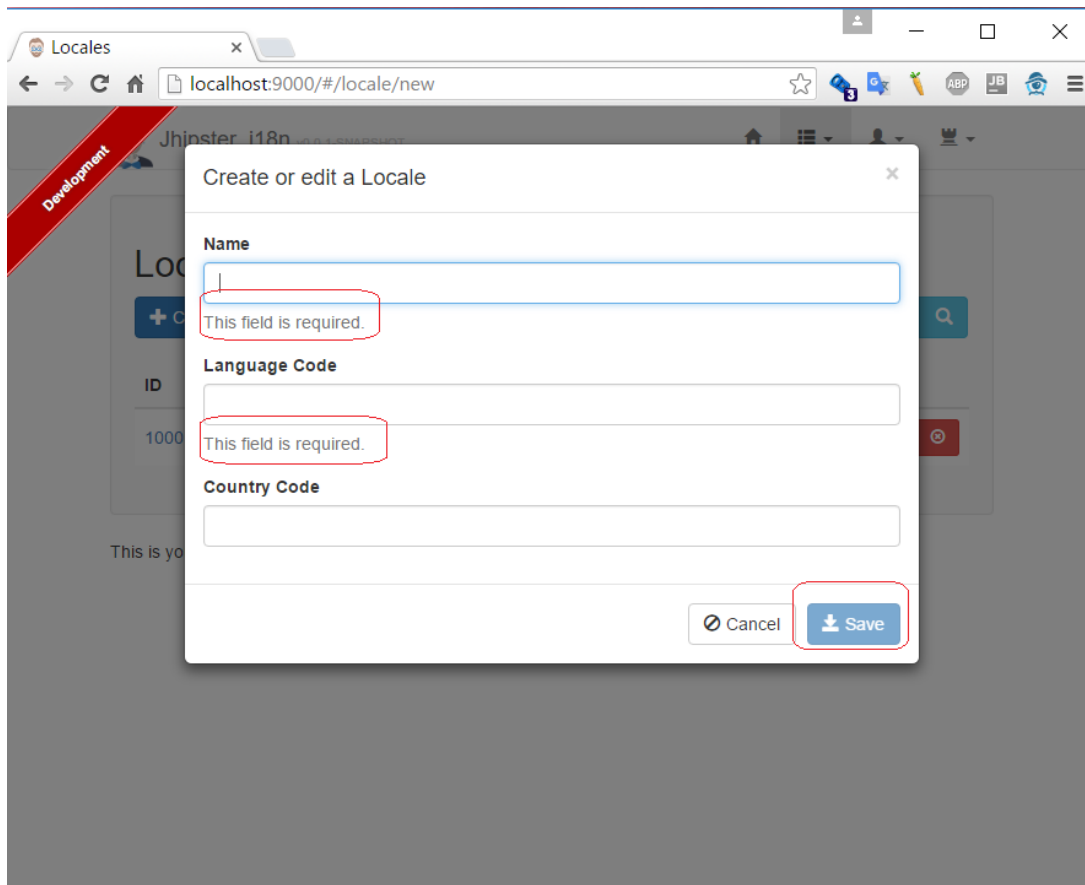
Figure 13: Form validation when creating a Locale

So what's the problem here? No problem. It's the lack of these features in ResourceBundle page when trying to create a ResourceBundle is the problem. This problem can be seen in Figure 12. There is nothing to stop user from submitting the form even when the required fields are empty. That is, no client side Javascript validation.

Culprit? Since all the code was generated off i18n.jh, going back to take a closer look, we see

```
...
entity Locale {
  name String required,
  languageCode String required,
  countryCode String
}
...

entity ResourceBundle {
 resourceBundleName String,
 description String maxlength(100)
 status ResourceBundleStatus
}

relationship ManyToOne {
  ResourceBundle{Locale} to Locale
}

relationship ManyToOne {
  ResourceBundle{Module} to Module
}
...
```

The locale entity has required keyword appended to the mandatory fields but not ResourceBundle entity since it's using many-to-one mapping. So mandatory fields validation were never genrated for Locale and Module in ResourceBundle.

The fix will actually require both both frontend and backend (i.e. JPA) changes since user could disable frontend validation by disabling or changing Javascript in browser. When that happens, the backend validation should kick in and return error messages as seen in Figure 14 below.
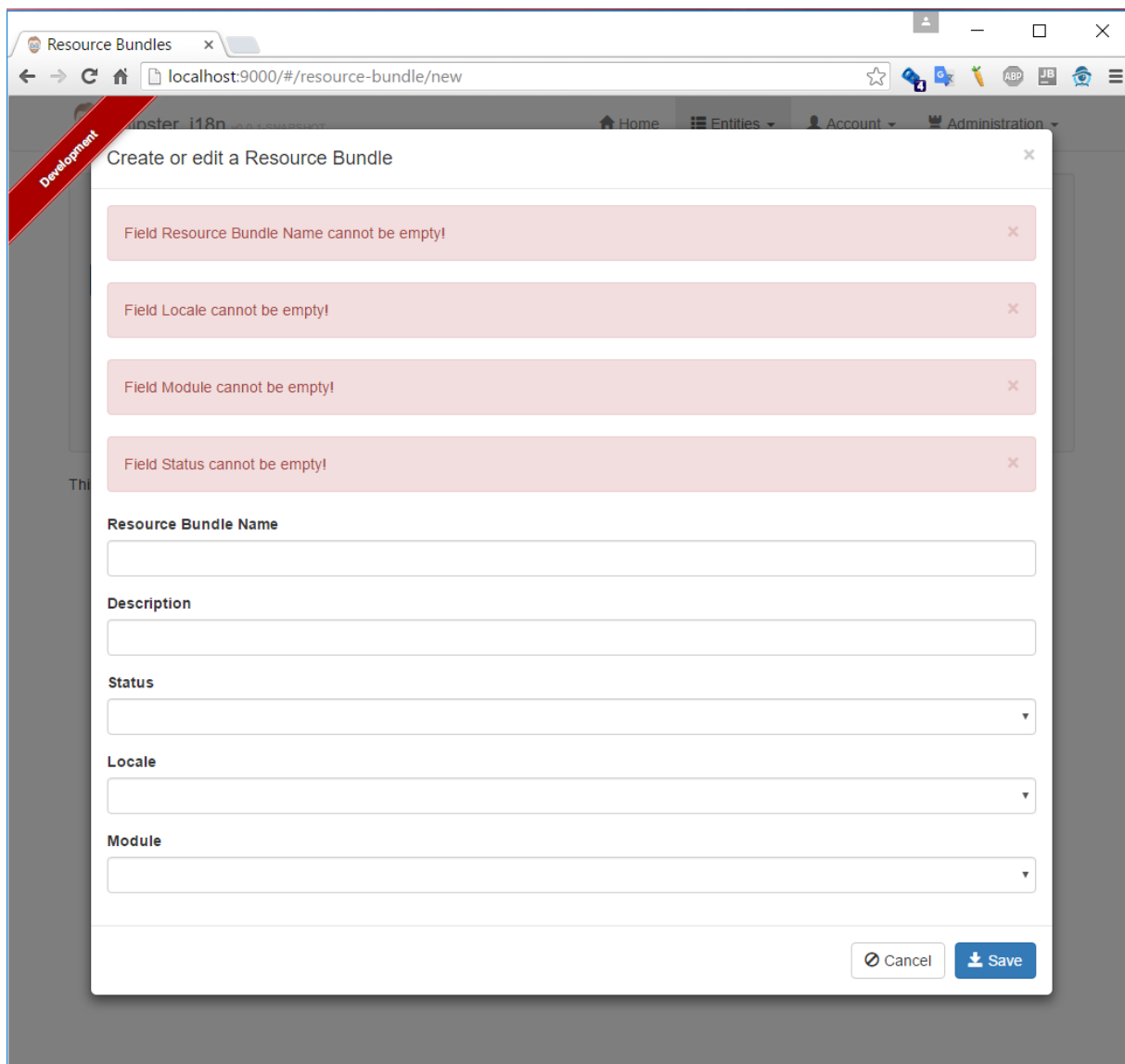
Figure 14: Error messages displayed by server side validations

The change requred for server side validation is actually quite simple in ResourceBundle JPA as highlighed red below

```
…
public class ResourceBundle implements Serializable {

    @NotNull
    @Column(name = "resource_bundle_name")
    private String resourceBundleName;
     …
    @Enumerated(EnumType.STRING)
    @Column(name = "status") @NotNull
    private ResourceBundleStatus status;

    @ManyToOne( fetch=FetchType.EAGER  )
    @JoinColumn(name="locale_id", nullable=false) @NotNull
    private Locale locale;
```

```
@ManyToOne( fetch=FetchType.EAGER )
@JoinColumn(name="module_id", nullable=false) @NotNull
private Module module;

...
```

So how does the @NotNull annotation resulted in browesr displaying error message? Goood question.

Checking console will reveals *org.springframework.web.bind.MethodArgumentNotValidException* is thrown at runtime when user submits the form. This exception is then caught by *ExceptionTranslator.processValidationError()* which returns ErroDTO back to browser in JSON format as

```
{
  "message": "error.validation",
  "description": null,
  "fieldErrors": [
    {
      "objectName": "resourceBundle",
      "field": "resourceBundleName",
      "message": "NotNull"
    },
    {
      "objectName": "resourceBundle",
      "field": "locale",
      "message": "NotNull"
    },
    {
      "objectName": "resourceBundle",
      "field": "module",
      "message": "NotNull"
    },
    {
      "objectName": "resourceBundle",
      "field": "status",
      "message": "NotNull"
    }
  ]
}
```

NotNull from above will result in the message template 'Field {{ fieldName }} cannot be empty!' retrieved from global.json in browser, i.e.

```
// In global.json, there exists a 'NotNull' entry
...
  "error": {
    ....
    "NotNull": "Field {{ fieldName }} cannot be empty!",
    ...
  },
  "footer": "This is your footer"
}
```

The placehoder *{{fieldName}}* will then be replaced by a value retrieved from resourceBundle.json (since objectName from JSON response is resourceBundle) using the value of field from JSON response.

Take the mandatory 'resource bundle name' for example. In JSON response listed above, we see

```
    …
    "fieldErrors": [
      {
         "objectName": "resourceBundle", // ➔ use resourceBundle.json to get placeholder value
         "field": "resourceBundleName",    // ➔ the key to retrieve placeholder value from resourceBundle.json
         "message": "NotNull" // ➔ the key to get 'message template' from global.json
      },
      {
  …
```

As clear as mud? Good. To return a custom error message, tweak data in *ErrorDTO* returned to browser in *ExceptionTranslator* and its matching JSON files.

This is only server side validation. What about client side? Well, things are about to get trippy so fasten your seat belt and get ready for some flashback in AngularJS form validations here, here, here and here.

Now take 'ResourceBundle Name' field as an example, the code required in resource-bundle-dialog.html to add frontend mandatory validation is highlighted red below

```
        <div class="form-group">
          <label class="control-label" translate="jhipsterI18NApp.resourceBundle.resourceBundleName"
               for="field_resourceBundleName">Resource Bundle Name</label>
          <input type="text" class="form-control" name="resourceBundleName" id="field_resourceBundleName"
               ng-model="vm.resourceBundle.resourceBundleName" required />
          <div ng-show="editForm.resourceBundleName.$invalid">
            <p class="help-block"
              ng-show="editForm.resourceBundleName.$error.required" translate="entity.validation.required">
              This field is required.
            </p>
          </div>
        </div>
```

Let's take this a bit further. User may also want to see the *red asterisk* after each required field's label. After quick StackOverflow meditation, we should have the following at the end of ***main.css***

```
        /* red asterisk for required fields */
        .required label:after {
          color: #e32;
          content: ' *';
          display:inline;
        }
```

To really show the red astersik, add the 'required' CSS class as highlighted red below for each required field

```
<div class="form-group required">
  <label class="control-label" translate="jhipsterI18NApp.resourceBundle.resourceBundleName"
      for="field_resourceBundleName">Resource Bundle Name</label>
  <input type="text" class="form-control" name="resourceBundleName" id="field_resourceBundleName"
      ng-model="vm.resourceBundle.resourceBundleName" required/>
    …
</div>
```

So we end up with Figure below. So far so good.
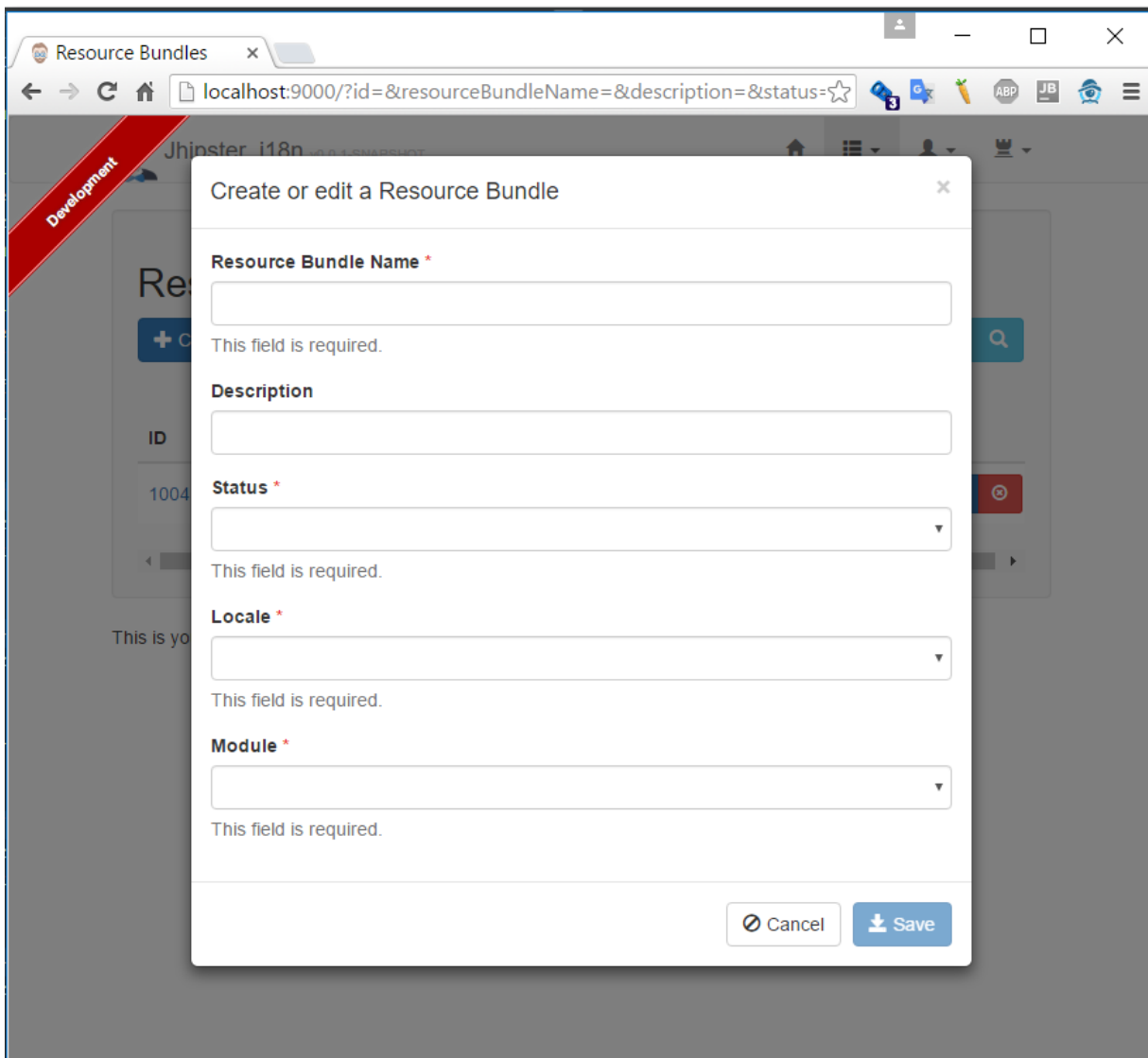
Figure 15: Resource Bundle dialog with upgraded form validation

What if user doesn't want to see the required message at first but only the red asterisk for required field? And the red required message would only be shown after submitting the form by clicking Save button provided the required field is still empty. In this case, the Save button won't be disabled at first but only disabled after failing validation by clicking Save button. After fixing the error, the Save button will be enabled once again.

Easy right? Let's pick the Locale entity for this picky client requirement. This would require us to introduce a JS variable named, say 'submitted' in the **Angular Controller** *locale-dialog.controller.js* backing *locale-dialog.html* to keep track of whether or not the form has been submitted. Then only show the mandatory message if the field is empty **AND** the form has been submitted.

Angular Controller is mentioned for the 1ˢᵗ time in this tutorial so for the uninitiated, see here, here and here. The required change in controller *locale-dialog.controller.js* is highlighted red below

```
function LocaleDialogController ($timeout, $scope, $stateParams, $uibModalInstance, entity, Locale) {
    var vm = this;
    ...

    function save () {
        vm.isSaving = true;
        vm.submitted = true;
        ...
    }

    function onSaveSuccess (result) {
        $scope.$emit('jhipsterI18NApp:localeUpdate', result);
        $uibModalInstance.close(result);
        vm.isSaving = false;
        vm.submitted = false;
    }
    ...
}
```

Then added the following code as highlighted in locale-dialog.html.

```
<form name="editForm" role="form" novalidate ng-submit="vm.save()" show-validation>
    ...
    <div class="modal-body">
        ...
        <div class="form-group required" ng-class="{'has-error': editForm.name.$invalid && vm.submitted}">
            <label class="control-label" translate="jhipsterI18NApp.locale.name" for="field_name">Name</label>
            <input type="text" class="form-control" name="name" id="field_name"
                ng-model="vm.locale.name"
                required />
            ...
        </div>
        <div class="form-group required" ng-class="{'has-error': editForm.languageCode.$invalid &&
vm.submitted}">
            <label class="control-label" translate="jhipsterI18NApp.locale.languageCode"
for="field_languageCode">Language Code</label>
            <input type="text" class="form-control" name="languageCode" id="field_languageCode"
                ng-model="vm.locale.languageCode"
                required />
            ...
        </div>
```

```
            …

        </div>
            …
        <button type="submit" ng-disabled="editForm.$invalid && vm.submitted" class="btn btn-primary">
            <span class="glyphicon glyphicon-save"></span> <span
    translate="entity.action.save">Save</span>
        </button>
        </div>
    </form>
```
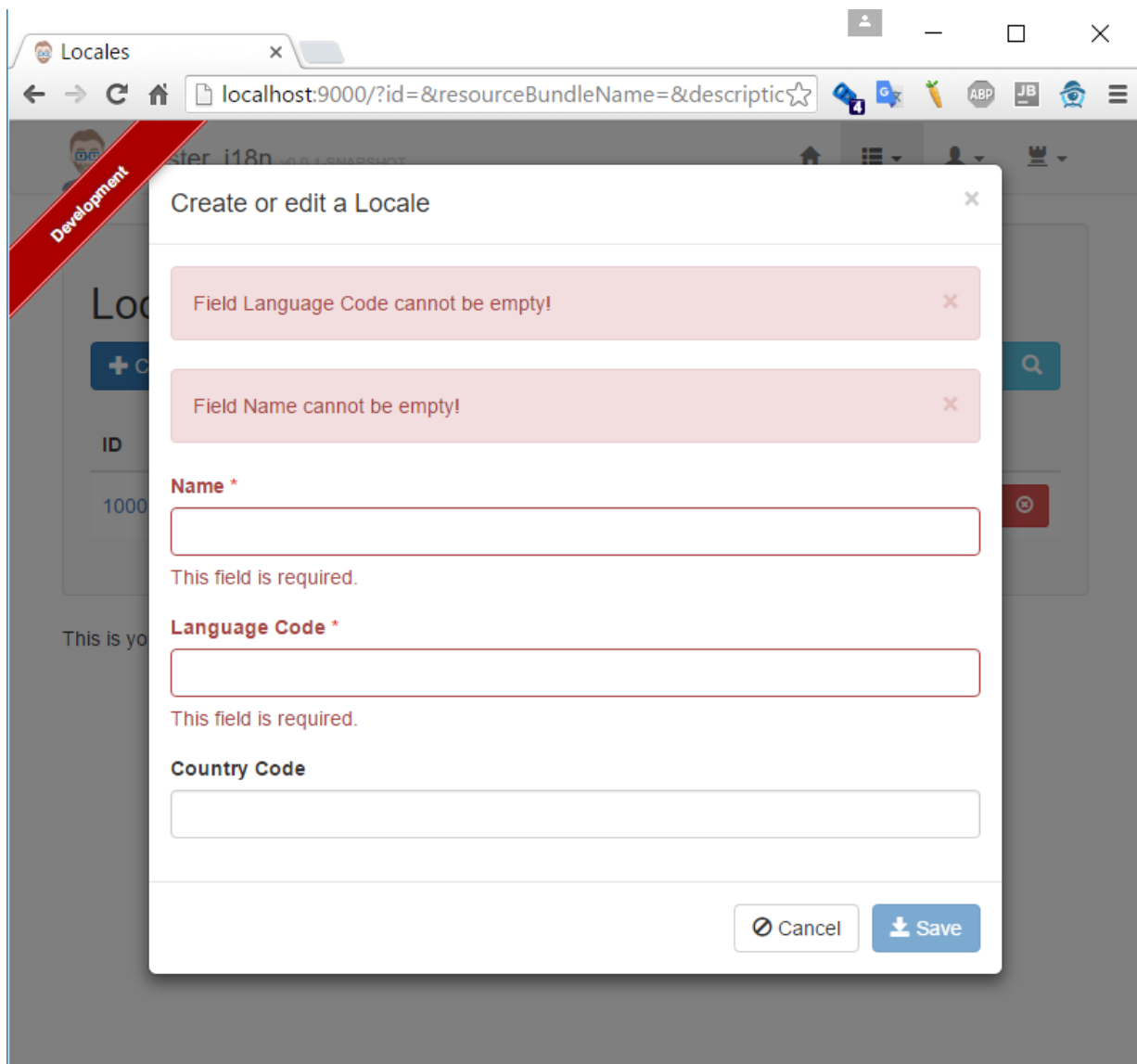
Then everything works as expected in Figure 16.

Figure 16: Clicking Save button when every field is empty. Both server and client side validations kick in.

This wraps up our AngularJS form validation tutorial.

See '*git checkout –f commit-10*' for code in this section.

Fixing database bug and adding more unique constraints to entities

By now if you have tried to add a KeyValue record through 'Create or edit a Key Value' dialog in browser, you may see internal server error message in alert box. Console will also show error message having something to do with the column resource_bundle_resource_bundle_id of key_value table. Checking actual key_value table in pgAdmin III reveals the shocking bug that there actually exists a column named *resource_bundle_resource_bundle_id* ! How did that get in there?

Remember in the Liquibase tutorials above, I endorse the practice of running gradle task 'liquibaseDiffChangelog' to generate Liquibase changelog XML file (containing database constraints declared in JPA) then reference the newly generated changelog file in master.xml so new database changes can be generated after dropping database and then restart server.

Well, it turns out this is a dangerous practice if you simply reference the generated changelog file in master.xml without scrutinizing what's actually in the changelog file. Apparently, a lot of unnecessary junk can be generated, while most seems harmless, a few will be fatal. In our case, the resource_bundle_id column in key_value table has been renamed to *resource_bundle_resource_bundle_id* in the changelog!!

So bear that in mind when using the gradle task 'liquibaseDiffChangelog'. I now run it to generate the database changlog XML file then delete all the crap before referencing it in master.xml. Or simply copy and paste the bits you want from generated changelog to an existing one used by master.xml

While I am fixing this, I also added new unique constraints in JPA as below
- Locale
    - The values of language_code and country_code columns will be unique, i.e. no two Locale records will have the same language_code and country_code
    - The value of name column will be unique

- ResourceBundle: the value of *resource_bundle_name* column will be unique

- Module: the value of *name* column will be unique

For the database changes to take effect, following the drill mentioned before multiple times. If any of these unique constraints are broken when adding a record in I18N app, the Data Integrity Violation error message will appear in alert box as in Figure 12 above.

See '*git checkout –f commit-11*' for the fix in this section

Prototype Resource Bundle dialog to manage Key Value data in a table

By default, the Resource Bundle dialog (i.e. see Figure 15) doesn't display Key Value records attached to it. A much better UI is to display all Key Value data attached to the current Resource Bundle here in a table and allow the creation/update/delete of each Key Value data. This design is much intuitive and saves user the trouble of creating Key Value data in a different page.

Let's use a wireframe tool like Balsamiq to mock up the screens because that seems to be the trendy thing to do during development in nowadays. After some playing around, we end up with the following *ultra-realistic* mock-up screen. Wow!

Figure 17: Final UI design for Resource Bundle dialog

Clicking the 'Create a Key Value' button will insert an empty row in the table with *Edit* button, which when clicked, will become three buttons (Save, Delete and Cancel) as seen in Figure 17. At this stage, the data in the table row will be editable. Clicking *Cancel* button will then convert data of that row back to read only.

Let's not worry about the actual functionalities of creating, updating and deleting KeyValue from database right now. But concentrate on getting the front-end right.

1. Key Value list table will display actual KeyValue data of ResourceBundle from database if any
2. Clicking Edit button will hide the button then show Save, Delete and Cancel buttons as in Figure 17. Note the text areas in Key, Value and Description columns are now editable.
3. Save and Delete buttons won't do anything for now.
4. Clicking Cancel button will hide all three Save, Delete and Cancel buttons then show the Edit button. Note the text areas in Key, Value and Description columns are protected now as in Figure 17.
5. Clicking 'Create a Key Value' button will insert an empty row into the Key Value list table with Edit button
6. Clicking Delete button for an empty row (i.e. one without Key Id) will delete the row from the table
7. 'Key Id' and Key columns of the table will be sortable by clicking on the column header
8. Protect Locale and Module fields as seen in Figure 17. Once a resource bundle is created, don't allow user to change those else suffer impact of data corruption
9. Make the 'Create or edit a Resource Bundle' dialog take up 80% of browser widthk. By default, the size of dialog if fixed. Making the browser larger won't affect its size! Dialog is looking bad as things are getting crowded here with Key Value list table. We want to take as much real estate of the screen as possible

Easy eh! Get ready for some Javascript acrobat.

Start with feature 1. Resource Bundle page needs access to all KeyValue data attached to it. Does it have those?

Now we are really going to dive deep into AngularJS and need to have a good understanding of how everything is wired to work together, so get ready to hurt your brain a little bit.

For each entity, there exists an **<entity>.stat.js** file (generated by JHipster) which basically acts as a router that defines multiple pages it can route the request to if a certain URL is matched. Each page is defined as a state containing multiple properties, e.g. URL, HTML template to serve the request, controller backing the HTML template, data to initialize before loading the page ….

In our case, it's ***resource-bundle.state.js*** and will look confusing to someone new to AngularJS so get ready for some astral travel to [$state](), [AngularJS State Management with ui-router](), [State Manager]() and maybe [AngularJS Routing Using UI-Router]().

So looking at *resource-bundle.state.js* again, we see *resource-bundle-dialog.html* gets mapped to the state with name '*resource-bundle.edit*' and is backed by the controller *ResourceBundleDialogController* which will contain the JS functions for *resource-bundle-dialog.html* to call. Searching the word 'ResourceBundleDialogController' in all JS will confirm it's defined in *resource-bundle-dialog.controller.js.*

```
…
// In resource-bundle.js
.state('resource-bundle.edit', {
    parent: 'resource-bundle',
    url: '/{id}/edit',
    data: {
        authorities: ['ROLE_USER']
    },
    onEnter: ['$stateParams', '$state', '$uibModal', function($stateParams, $state, $uibModal) {
        $uibModal.open({
            templateUrl: 'app/entities/resource-bundle/resource-bundle-dialog.html',
            controller: 'ResourceBundleDialogController',
            controllerAs: 'vm',
            backdrop: 'static',
            size: 'lg',
            resolve: {
                entity: ['ResourceBundle', function(ResourceBundle) {
                    return ResourceBundle.get({id : $stateParams.id}).$promise;
                }]
            }
        }).result.then(function() {
            $state.go('resource-bundle', null, { reload: true });
        }, function() {
            $state.go('^');
        });
    }]
})
…
```

We can also see before the page *resource-bundle-dialog.html* is served, a ResourceBundle record will be loaded from backend using the [Angular Service](#) *ResourceBundle* defined in **resource-bundle.service.js** The Angular Service *ResourceBundle* basically gets injected into resource-bundle-dialog.controller.js to call server-side restful service for CRUD operations. It's exactly like how a Spring bean in Java backend can be auto-wired to a service layer or controller class.

Coming back to the original question. Does *resource-bundle-dialog.html* have access to KeyValue data? Ehhh… No! Nowhere in resource-bundle-dialog.controller.js can we see the Angular service *KeyValue* (defined in key-value.service.js) injected in to access KeyValue data from backend.

Remember that in i18n.jh, a Many-To-One mapping was defined from KeyValue to ResourceBundle but not the other way around so a ResourceBundle record retrieved from backend knows nothing about its KeyValue data by default.

What we really need is a restful service implemented on server side (i.e. Java) and an Angular Service on the client side to call the server-side restful service that would return a list of KeyValue records attached to the same Resource Bundle. You can see things are starting to get interesting (i.e. complicated).

So let's create the server-side restful service first. First we need to create a service that returns a list of KeyValue objects using resource bundle id as input argument. How difficult is it? Unfortunately, it's dead simple by using the power of Spring Data JPA @Query annotation. Good place to do this is KeyValueRepository as highlighted red below.

```java
@Transactional(readOnly = true)
public interface KeyValueRepository extends JpaRepository<KeyValue,Long> {

    @Query( value  = "select kv from KeyValue kv where kv.resourceBundle.id  = ?1")
    List<KeyValue> getKeyValuesByResourceBundleId(Long id);
}
```

Compared with JPA 2, this is way simpler. For those needing a refresher on @Transactional annotation, see here and here. Now we create the actual restful service by the following simple code in KeyValueResource. Almost too easy but again simplicity is the best form of sophistication.

```java
@Timed
@RequestMapping(value = "/key-values/rb/{id}",
     method = RequestMethod.GET,
     produces = MediaType.APPLICATION_JSON_VALUE)
public List<KeyValue> getKeyValuesByResourceBundleId(@PathVariable Long id) {
    log.debug("REST request to get all KeyValues by Resource Bundle Id");
    List<KeyValue> keyValues = keyValueRepository.getKeyValuesByResourceBundleId( id );
    return keyValues;
}
```

All the hard work is done by *@RequestMapping* and *@PathVariable* so make sure one has a solid understanding of it here, here. Hey, where is unit test to prove it work? Umm … I am leaving that as an excellent exercise for the reader ;).

Now the restful service client in AngularJS. Angular Service will be used to create the restful client as the service can be passed around and injected into any other Angular controllers wanting to use it as a singleton. Inside the service, the most common Angular way to call a restful service will be to use the **$resource** object (here, here) so let's create an Angular Service named *KeyValueSearchBy_RB_Id* in the file *key-value.get-by-resourceBundle-id.js*

```javascript
(function() { 'use strict';
  angular.module('jhipsterI18NApp').factory('KeyValueSearchBy_RB_Id', KeyValueSearchBy_RB_Id);

  KeyValueSearchBy_RB_Id.$inject = ['$resource'];
  function KeyValueSearchBy_RB_Id( $resource ) {
    var resourceUrl =  'api/key-values/rb/:id';
    return $resource(resourceUrl, {}, {
      'query': { method: 'GET', isArray: true}
    });
}}) ();
```

To use it in our ResourceBundleDialogController, simply added the following code highlighted red in resource-bundle-dialog. controller.js

```
ResourceBundleDialogController.$inject = ['$timeout', '$scope', '$stateParams', '$uibModalInstance', 'entity',
    'ResourceBundle', 'Locale', 'Module', 'KeyValueSearchBy_RB_Id'];

function ResourceBundleDialogController ($timeout, $scope, $stateParams, $uibModalInstance, entity,
    ResourceBundle, Locale, Module, KeyValueSearchBy_RB_Id) {

    var vm = this;

    vm.resourceBundle = entity;
    vm.clear = clear;
    vm.save = save;

    if ( vm.resourceBundle.id === null ) {
        vm.locales = Locale.query();
        vm.modules = Module.query();
    } else {
        vm.keyValues = KeyValueSearchBy_RB_Id.query( {id : $stateParams.id} );
    }
```

Now turn out attention to the HTML file *resource-bundle-dialog.html* that will be using the **vm.keyValues** data. Now since I am too lazy to explain all the details the reader will have to refer to the Git commit below for those. And I will just highlight the things that I think is important.

The Key Value table will use ng-repeat directive to loop over vm.keyValues data from controller to render all Key Value data.

Feature 5 (i.e. inserting a new Key Value row) will be done by calling the JS function *addKeyValueRow()*added in controller as below. The key is the JS push function.

```
vm.addKeyValueRow = function( ) {
    vm.keyValues.push( {} );
}
```

Feature 6 (i.e. deleting a Key Value table row) will be done by calling the JS function *removeKeyValueRow()* added in controller as below. The magic is in the JS splice function.

```
vm.removeKeyValueRow = function( keyValue ) {
    var removeIndex = vm.keyValues.indexOf( keyValue );
    vm.keyValues.splice( removeIndex, 1);
}
```

Feature 7 is about sorting data by table header. This is achieved by using Angular filter orderBy. Note this is *client side* sorting. People who are interested in *server side sorting* using Spring Data can take a look at this.

```
…
<table class="table table-bordered table-hover" >
  <thead>
    <tr>
      <th><a href="" ng-click="orderByField='id'; reverseSort = !reverseSort">Key Id</a>
      </th>
      <th>
        <a href="" ng-click="orderByField='propertyValue'; reverseSort = !reverseSort">Key</a>
      </th>
      <th>Value</th>
      <th>Description</th>
      <th class="col-md-2">Actions</th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat="keyValue in vm.keyValues | orderBy:orderByField:reverseSort"
      ng-init="keyValue.isEditing = false; keyValue.saveBtnClicked = false" >
        …
```

Feature 9 makes the width of dialog taking up 80% of browser width. You may think it's easy. All you have to do is using CSS to tweak its size. Which one and how? After some digging, thanks to this StackOverflow post , the simplest way I found is adding the following red line in *resource-bundle-state.js*

```
…
.state('resource-bundle.edit', {
  …
  onEnter: ['$stateParams', '$state', '$uibModal', function($stateParams, $state, $uibModal) {
    $uibModal.open({
      templateUrl: 'app/entities/resource-bundle/resource-bundle-dialog.html',
      …
      size: 'lg',
      windowClass: 'app-modal-window',
    ….
  }]
})
…
```

Then simply add the following inline CSS in *resource-bundle-dialog.html* as (it could be put in main.css for share by every page)

```
<style type="text/css">
  .app-modal-window .modal-dialog {
    width: 80% !important;
  }
</style>
```

For the rest details, you will just have to take my word for it or check the Git commit below.

◆ See '*git checkout –f commit-12*' for the fix in this section

Implement Create, Update and Delete functions of Key Value to backend

The next step is to make sure the Save and Delete buttons of Key Value list table can actually create/update and delete Key Value data from backend respectively.

Delete function. Looking at the Delete button of Key Value list table in Figure 17, do we really want to delete Key Value record data from database straight away if user clicks on the button by accident? Of course not. We should pop up a 'Confirm delete operation' dialog as in Figure 18 below.

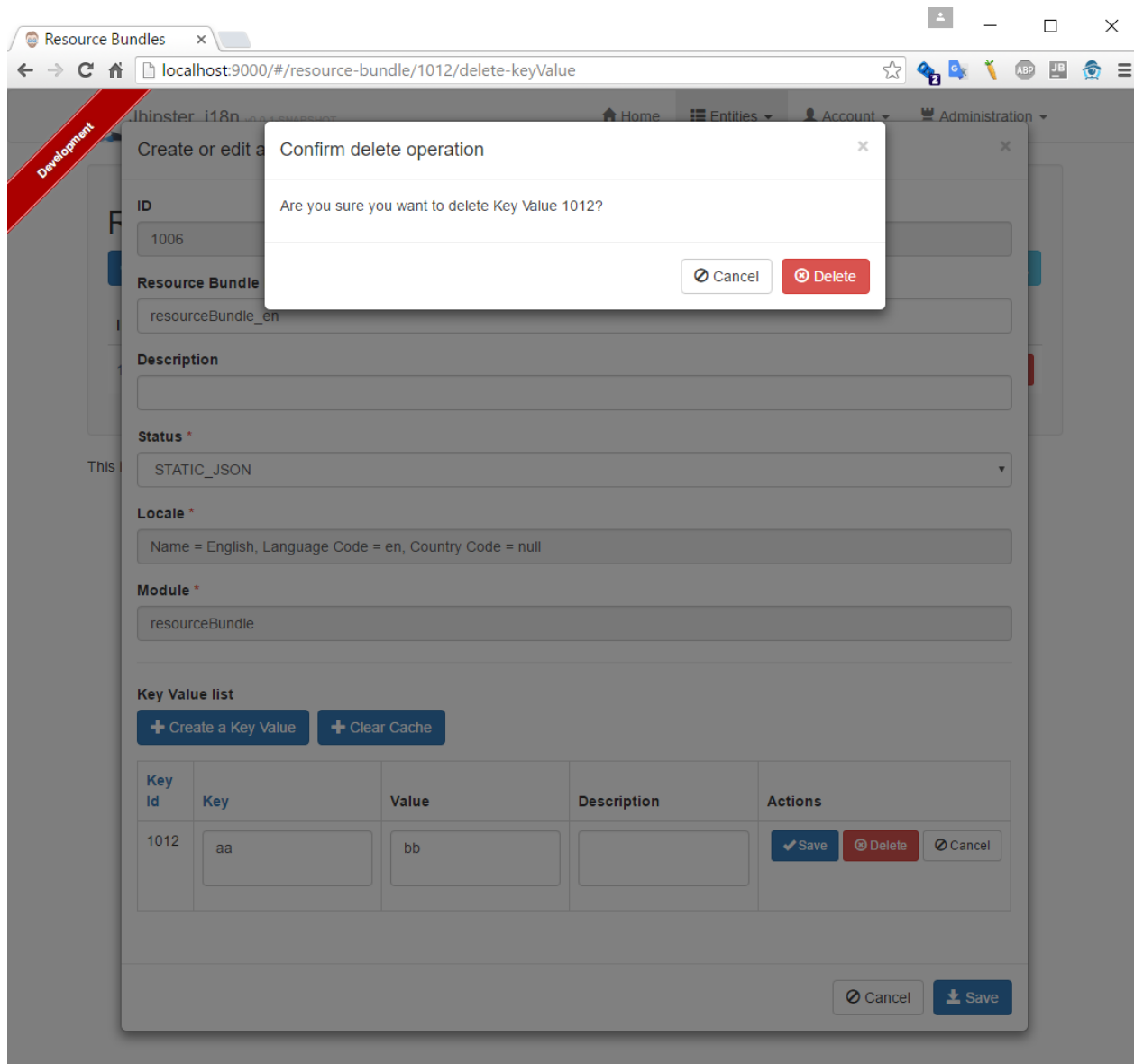
Figure 18: Clicking Delete button will show 'Confirm delete operations' dialog.

If the Delete button is clicked on a table row without Key Id, the the row should be removed straight away since the Key Value hasn't been persisted to databse yet.
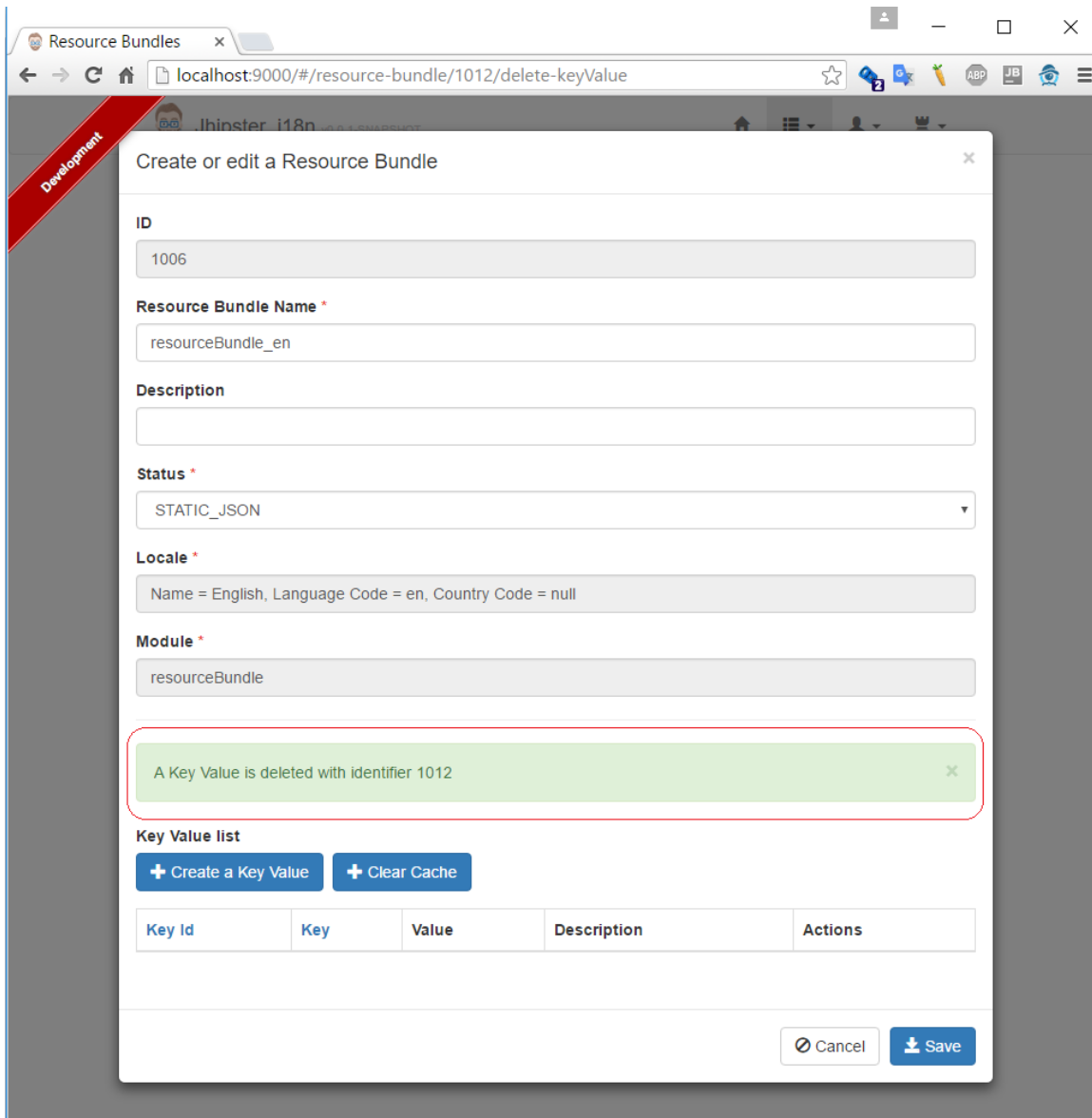
Figure 19: Successful delete message in green alert box after clicking the the Delete button from dialog in Figure 18.

Also we would want to add an alert message box just above the Key Value list table after deleting the Key Value successfully by clicking the Delete button from the confirmation dialog. See Figure 19. The red alert box with error message should be displayed instead if the delete wasn't successful from backend.

To implement the Confirm delete dialog, we turn our attention to the Key Value pages generated by JHipster. This feature is implemented already out of the box by JHipster. After some digging, we should track down the actual HTML page used to implement the Confirm Delete dialog is *key-value-delete-dialog.html*. The code specifying the use of this HTML page is found in *key-value.state.js* as below

```
.state('key-value.delete', {
    parent: 'key-value',
    url: '/{id}/delete',
    data: {
        authorities: ['ROLE_USER']
```

```
        },
        onEnter: ['$stateParams', '$state', '$uibModal', function($stateParams, $state, $uibModal) {
          $uibModal.open({
            templateUrl: 'app/entities/key-value/key-value-delete-dialog.html',
            controller: 'KeyValueDeleteController',
            controllerAs: 'vm',
            size: 'md',
            resolve: {
              entity: ['KeyValue', function(KeyValue) {
                return KeyValue.get({id : $stateParams.id}).$promise;
              }]
            }
          }).result.then(function() {
            $state.go('key-value', null, { reload: true });
          }, function() {
            $state.go('^');
          });
        }]
      });
```

So what do we do? Copy and paste that code and adapt it for use in our *resource-bundle.state.js* of course. Everything is almost the same except for the code highlighted red

```
        // Showing 'Confirm delete operation' dialog for deleting Key Value record
        .state('resource-bundle-edit-delete-key-value', {
          parent: 'resource-bundle',
          url: '/{id}/delete-keyValue',
          data: {
            authorities: ['ROLE_USER']
          },
          onEnter: ['$stateParams', '$state', '$uibModal', function($stateParams, $state, $uibModal) {
            $uibModal.open({
              templateUrl: 'app/entities/key-value/key-value-delete-dialog.html',
              controller: 'KeyValueDeleteController',
              controllerAs: 'vm',
              size: 'md',
              resolve: {
                entity: ['KeyValue', function(KeyValue) {
                  return KeyValue.get({id : $stateParams.id}).$promisse;
                }]
              }
            }).result.then(function() {
            }, function() {
              // required else clicking the Delete button using ui-sref attribute to enter this state again
              // to open the 'Confirm Delete' dialog won't work
              $state.go('^');
            });
          }]})
```

Now we use the state name *resource-bundle-edit-delete-key-value* in the Delete button of resource-bundle-dialog.html as

```
 
<button type="button" class="btn btn-danger btn-sm" ng-show="keyValue.isEditing && keyValue.id"
    ui-sref="resource-bundle-edit-delete-key-value({id:keyValue.id})">
    <span class="glyphicon glyphicon-remove-circle"></span>
    <span class="hidden-xs hidden-sm" translate="entity.action.delete">Delete</span>
</button>

<button type="button" class="btn btn-danger btn-sm" ng-show="keyValue.isEditing && !keyValue.id"
    ng-click = "vm.removeKeyValueRow( keyValue )" >
    <span class="glyphicon glyphicon-remove-circle"></span>
    <span class="hidden-xs hidden-sm" translate="entity.action.delete">Delete</span>
</button>
 
```

Note there are two Delete buttons in the code above. First is used for deleting Key Value that has been persisted to database already. The second Delete button is for deleting Key Value that hasn't been persisted yet. At anyone time, only one Delete button will be visible, depending on whether the key value has non-empty primeary key, i.e. keyValue.id.

```
<div ng-show="!vm.alertForResourceBundle">
    <jhi-alert-error></jhi-alert-error>
    <jhi-alert></jhi-alert>
</div>
```

What is vm.alertForResourceBundle? It's a JS variable configured in *resource-bundle-dialog.controller.js* to decide if the alert box for Resource Bundle or Key Value operation will show. Remember we also an alert box for Resource Bundle. See figure 16. Without this control, both alert boxes will show together.

In resource-bundle-dialog.controller.js, we configure *vm.alertForResourceBundle* as

```
// called when Save button of Key Value is clicked
vm.saveKeyValue = function( keyValue ) {
    keyValue.saveBtnClicked = true;
    vm.alertForResourceBundle = false;
    ...
};

// called when Save button of Resource Bundle is clicked
function save () {
    vm.isSaving = true;
    vm.submitted = true;
    vm.alertForResourceBundle = true;
    ...
}
```

 So now everything is working as expected in this section. See '*git checkout –f commit-13*' for details.

54

Serve I18N JSON data through database and caching it

Now we come to the final and most important part of I18N app. All the previous work is just preparation for this – doing internationalization with database by serving JSON files through restful service.

Beforing moving to the final act, let's fix one JS bug regarding the alert box first. One unexpected side effect of showing the info message in the alert box (i.e. Figure 19) for 'Create or edit a Resource Bundle' dialog (or any dialog) is that the alert box with the same info message will also show up in the parent page of the dialog once the dialog is closed. See Figure 20.
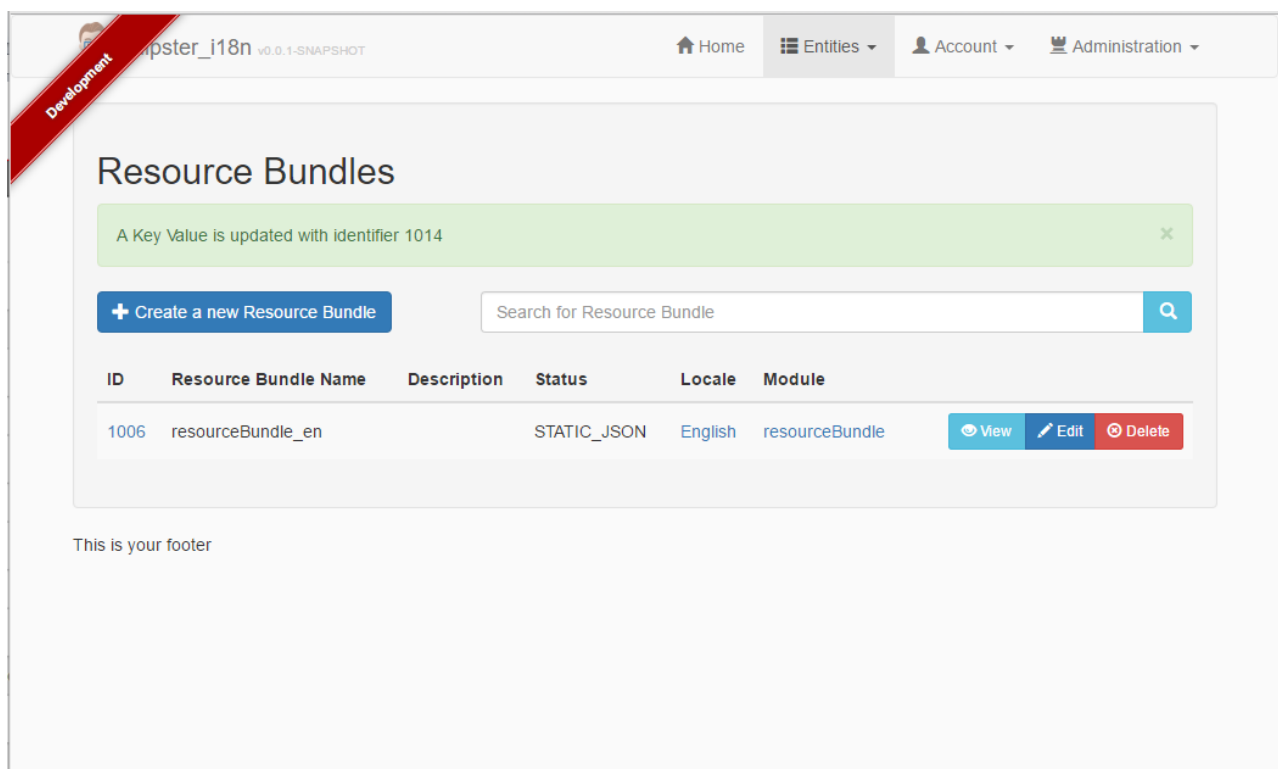

Figure 20: Whatever message displayed by the alert box in dialog will also be displayed in the page that opens the dialog once the dialog is closed.

We know the alert box is implemented by using <jhi-alert></jhi-alert> component in HTML. This component is in turn implemented in alert.directive.js which has code like

```
function jhiAlertController($scope, AlertService) {
    var vm = this;
    vm.alerts = AlertService.get();
    $scope.$on('$destroy', function () {
            vm.alerts = [];
    });
}
```

After putting breakpoint on this function by using Chrome Developer Tool(i.e. F12 -> Sources) and some testing, it leads to the Aha moment that whenever the dialog is closed the $destroy event gets fired

somewhere then the event is caught in the event listener *$scope.$on*. In case anyone is hazy with event handling in Augular, see [here](#) and [here](#).

So the simple fix I can see is to simply create a method that will close all alert boxes in AlertService then call that method within *$scope.$on* as highlighted red below

```
// in alert.directive.js
…
function jhiAlertController($scope, AlertService) {
   var vm = this;
   vm.alerts = AlertService.get();
   $scope.$on('$destroy', function () {
         AlertService.closeAllAlerts();
         vm.alerts = [];
   });
}
…

// in alert.service.js
…
function getService ($timeout, $sce,$translate) {
  …
  return {
     factory: factory,
     isToast: isToast,
     add: addAlert,
     closeAlert: closeAlert,
     …
  };
  …
  // Close all alerts
  function closeAllAlerts() {
     for ( alert in alerts ) {
        closeAlert( alert.id, alerts )
     }
  }
```

Now some functinal design for the final act.

1. A single restful service will be created to serve the JSON files such that depending on the URL requested, different ResourceBundle can be retrieved from backend to serve the request => no fixed URL for this service

2. The restful service from 1 needs to cache requested ResourceBundle so the cached data can be used to serve future requests of the same ResourceBundle without accessing database again

3. A 'Clear Cache' mechanism needs to be implemented so the 'Clear Cache' button from Figure 19 can be clicked to clear all the cached data of current ResourceBundle after real time changes to Key Value data attached to it.

4. A new Restful service needs to be implemented in a non-intrusive way so
   o No changes to Angular frontend is required
   o Existing static JSON files can be migrated one by one without impacting the running server

For feature 1, a new restful controller class will need to be created to serve URL with the following format so as not to be intrusive (i.e. feature 4).

http://{domain-name}:port/i18n/{language-code}/{module-name}.json

This means the URI needs to start with '/i18n'. It's used by AngularJS to access static JSON files so we don't change it to minimize impact to frontend. The static JSON file name needs to map to *{module-name}* so we use the *Name* field of 'Create or edit a Module' dialog to define this {module-name}. See Figure 21 below. Note the case of the Name has to match that of actual JSON file name, i.e. global is not the same as Global.
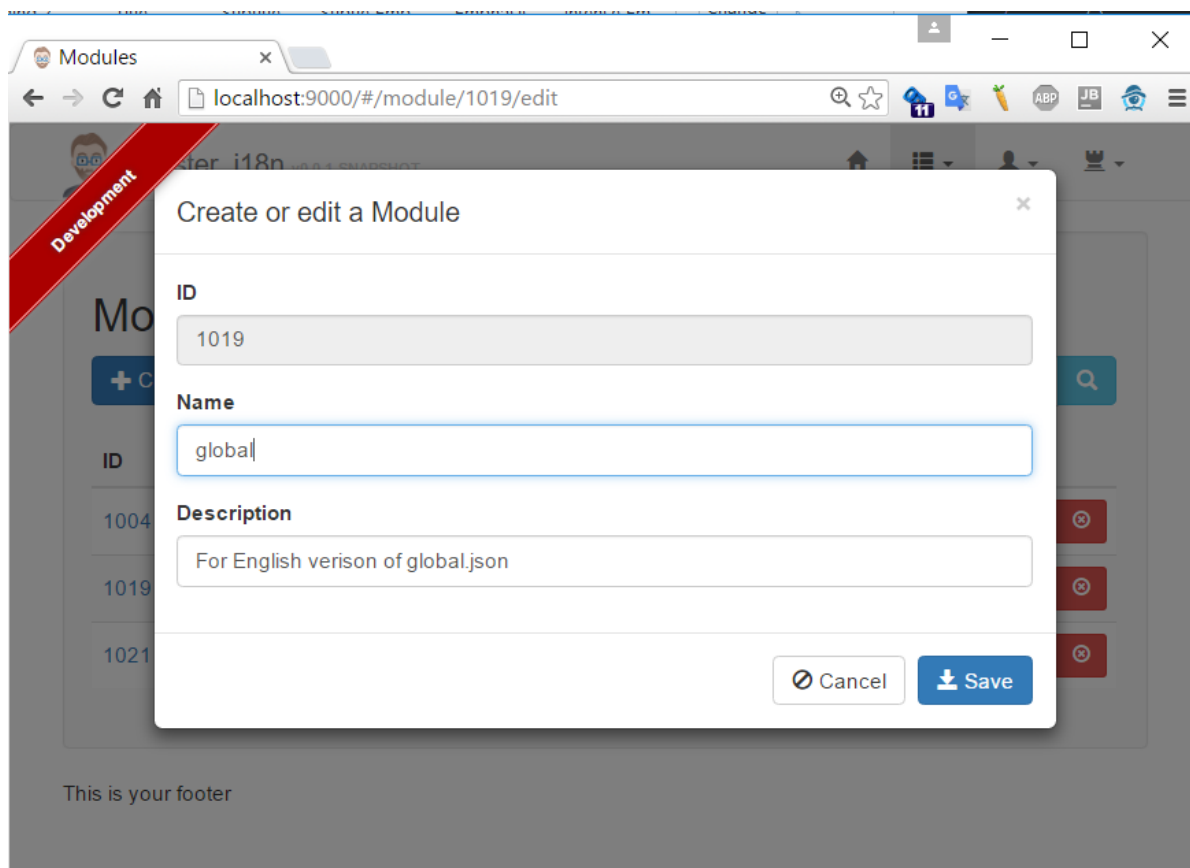

Figure 21: The Name field specifies the *{module-name}* of URI

Now we implement a new Rest controller class as below

```
@RestController
@RequestMapping("/i18n")
public class I18nController {

    @Inject
    private I18nService i18nService;

    @RequestMapping(value = "/{languageCode}/{moduleName}.json",
        method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
    @Timed
    public ResponseEntity<String> getI18nJson(HttpServletRequest request, @PathVariable String
        languageCode, @PathVariable String moduleName) throws URISyntaxException,
        MalformedURLException, JsonProcessingException {
      ...
    }
    ...
}
```

Note the URI starting with '/i18n' requirement is satisfied by @RequestMapping("/i18n") and the dynamic mappings of {language-code} and {module-name} are taken care of by other red, highlighted code.

For feature 2, a Spring Service component (i.e. class annotated with @Service) class named *I18nService* will be created and injected into the restful controller class which can delegate actual work of caching data by calling methods of the injected Spring service.

For feature 3, we simple implement a clearCache(..) method in the *I18nService* class above and call it from a new restful service method (serving request with URI */resource-bundles/clear-cache*) in ResourceBundleResource as

```
/**
 * POST  /resource-bundles/:id/clear-cache : clear the key-value map cache for the "id" resourceBundle
 *
 * @param resourceBundle   the ResourceBundle to clear key-value map cache
 * @return the ResponseEntity with status 200 (OK)
 */
@RequestMapping(value = "/resource-bundles/clear-cache",
    method = RequestMethod.PUT,
    produces = MediaType.APPLICATION_JSON_VALUE)
@Timed
public ResponseEntity<Void> clearCache(@RequestBody ResourceBundle resourceBundle) {
    log.debug("REST request to clear cache for ResourceBundle: {}", resourceBundle.getId());
    i18nService.clearCache( resourceBundle );
    return ResponseEntity.ok().headers(HeaderUtil.createEntityDeletionAlert("resourceBundle",
        "CLEAR_CACHE_SUCCESS " )).build();
}
```

For feature 4, remember ResourceBundle table was generated with a status field having only one of three possible values: DISABLED, DATABASE _JSON, STATIC_ JSON. This is how feature 4 will be implemented. Note also that this status field is configurable per Resource Bundle basis in browser as seen in Figure 19.

When the status is DISABLED, nothing will be returned so the labels served by that ResourceBundle in HTML will be replaced by the keys used to access the lables in JSON file as in Figure 22 below.This is also a good way for testing I18N at runtime to see what key maps to what label.
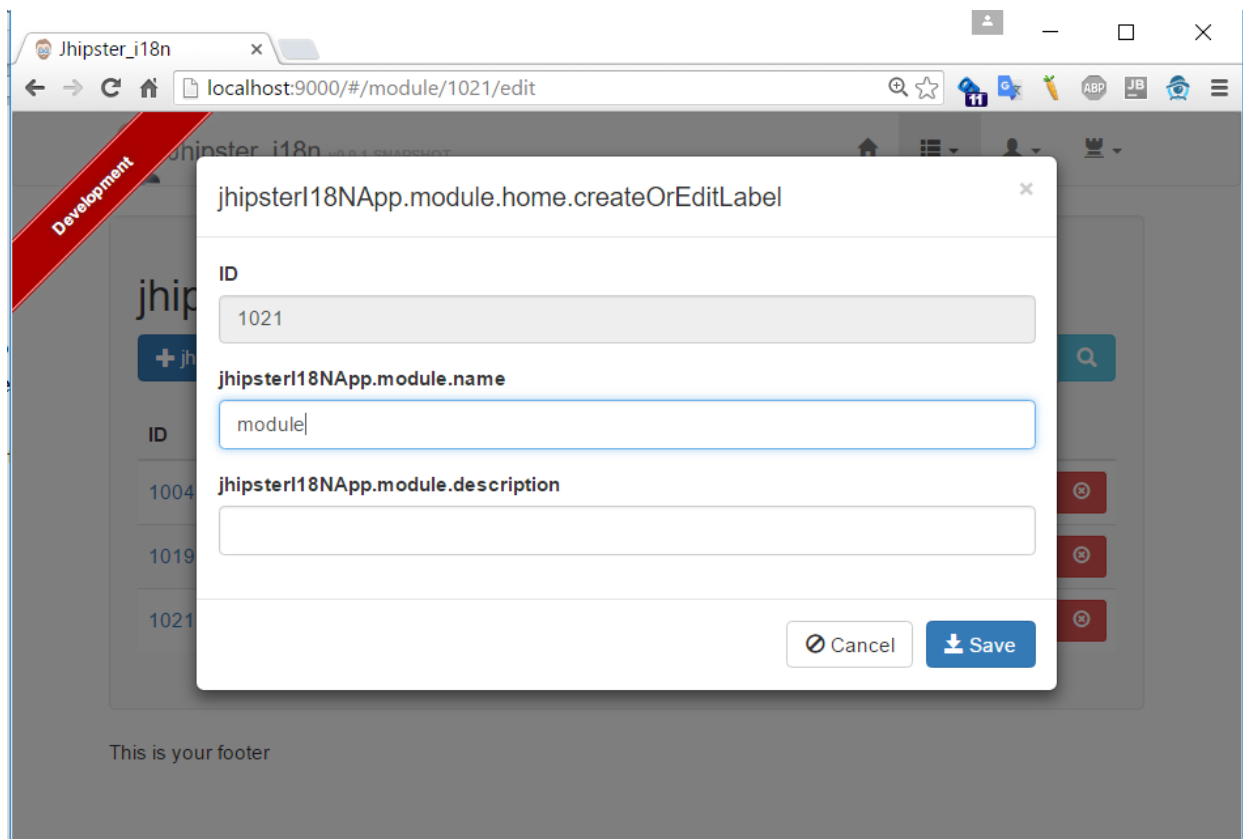

Figure 22: Only the keys used to access labels in JSON will show once the status of ResourceBundle is changed to 'DISABLED'

One caveat of changing 'status' of ResourceBundle or clearing cache through 'Clear Cache' button in 'Create or edit a Resource Bundle' dialog is that since browser also caches the JSON files received, to see the changes, one may need to refresh the browser by pressing F5.

When status is STATIC_JSON, the request will be served by using static JSON file at a different URI */i18n/json/{language-code}/{module-name}.json*. Note the red '/json' is what's new. To make this happen, we need to cut and paste all existing JSON files from the folder */jhipster_i18n/src/main/webapp /i18n/en/* to */jhipster_i18n/src/main/webapp /i18n/json/en/*.

When status is DATABASE_JSON, *I18nController* will delegate actual caching to *I18nService*. Easy, job done! See '*git checkout –f commit-14*' for code in this section.

This concludes the whole tutorial. Hopefully the reader is one step closer to becoming a Hipster. ☺