LAVA Development Guide

UCSF Memory and Aging Center

Document Revision Date: 11/26/13

Overview and Purpose 4
LAVA Development Environment Installation 6
- MySQL 6
- Java 7
- Eclipse 7
- Apache Ant 8
- Apache Tomcat Web Application Server 8
- Install a JDBC connector for MySQL in Tomcat. 10
- Install the Java Transaction API (JTA) 10
- Configure Logging 10
- Create Directory for Context Descriptor 11
- Configuring Eclipse to Launch Tomcat 12
- Start Server 13
- Test your installation 13

LAVA Source Code Repository 14
- Configuring LAVA CVS Repository in Eclipse 14
- Obtaining LAVA source code 15

LAVA Relational Database 17
- Schema 17
- Running Complete Database Script 18
- Generating Complete Database Script 19
- Metadata / List Data 19

Building LAVA Applications 22
- Eclipse LAVA Java configuration 22
- Eclipse LAVA Projects 25
- Build Targets 26
- build.properties 27
- Eclipse Build Configurations 29
- Tomcat Application Context Descriptor 30
- Development Cycle 31
- Metadata 32
- Data Reload Shortcuts 34

Authentication 36

Users 36
Groups 37
Authorization 38
Roles 38
Permissions 39
Logging 40
Java Debugging 41
Production Release 42
WAR file creation 42
Initial Release 42
Database Schema Updates 42
Metadata Updates 42
SSL 43
Creating a New App from the LAVA Demo App 44
LAVA Framework 51
System Architecture and Requirements 51
Architectural Highlights: 52
Optimal / Minimum System Requirements 52
Licensing 52
Design Philosophy 53
Logical Structure 53
Core Scope 55
Action 55
Admin 55
Auditing 55
Authorization 55
Dao 55
List 55
Metadata 56
Reporting 56
Session 56
Webflow 56
CRMS Scope 56

Patient / Project Context (not a module but important to explain here) 56
Assessment 57
Enrollment 57
People 57
Project 57
Scheduling 57
Development of Applications with LAVA CRMS 57
Determining Where Application Functionality Fits in LAVA CRMS 57
Domain Model Objects 58
Action Definitions 58
(Page) Flow Types 59
Form Actions and Component Handlers 62
View Layer 62
Reports 63
Some LAVA Terminology Defined 64
Extending and Customizing 66
Instance and Scope 66
Customizing the Interface 66
Color Scheme 66
Logo 67
Login Pages 67
Welcome Page 67
Footer 67
Metadata 67
Application Specific Properties 71
Property Metadata Override 72
Customizing Lists Based on Project 72
Customizing Enrollment Statuses 74
Defining Modules (Tabs) and Sections (Sub-tabs) 76
Decorators 76
Module Tabs 77
Section Sub-tabs 78
Default Actions 78

Customizing Action and Reports Panel 80
Skip Logic 81
List Definition 81
Deidentified Patient Record 83
Customizing Entities and Actions 83
Instance Customization 85
Patient Entity Instance Customization 86
Visit and EnrollmentStatus Entity Instance Customization 89
Entity Dynamic Customization 93
Visit and EnrollmentStatus Entity Dynamic Customization 95
Entity Prototype Default Values 98
Instruments 99
Instrument Creation 99
Streamlined Creation Scripts 104
LAVA Error Codes 111
Standard Error Codes 111
Skip Error Codes 111
Instrument Skip Logic 112
Creating Visit Instrument Prototypes 112
Calculations 114
Data Double Entry 114
Data Review 115
Customizing Instrument Status 115
Customizing NACC UDS Patient ID 116

Overview and Purpose

LAVA is an open source web application framework developed at the UCSF Memory and Aging Center (MAC). LAVA was initially developed as an application for clinical researchers to centrally manage project administration and data collection for multiple related research protocols. Over time, the LAVA framework has been refined into both a general purpose web application framework and a specifically purposed clinical research management solution.

By describing LAVA as a web application framework we simply mean that LAVA provides one approach to providing the core services and functionality required for web application development. These services are built upon standard open source libraries and lower level web application frameworks.

One objective for freely sharing the LAVA framework is to give a jump start to other application developers (particularly those that work in the context of academic research), enabling them to develop web applications more quickly and with less expense.

Another, more complex, objective is the development of a community of academic research developers sharing a common platform; with the goals of learning from other developers and incrementally improving the quality and sophistication of the tools available to the community for developing research-oriented software solutions.

This is not to exclude other developers from adopting LAVA as a development environment, but rather to simply describe where we, as the originators of the LAVA platform, are coming from, and where we plan to focus our efforts in our future development of LAVA.

LAVA Development Environment Installation

This section covers installation of everything required to develop LAVA CRMS (Clincal Research Management System) applications, as well as applications built on the LAVA core infrastructure which may not be related to clinical research. It details the installation of each underlying technical component, without getting into anything specific to LAVA. Configuring these components to build LAVA applications is covered in the following section.

Because LAVA is Java based, the development environment can run on Linux, Mac OS X or Windows.

While Java is required to develop and run LAVA applications, the rest of the components covered in this section represent the recommended and current state of LAVA development, but there are alternatives:

- LAVA uses a relational database for persistent storage of data. LAVA can be used with many relational databases, as long as they are supported by the Hibernate ORM framework. However, currently, only MySQL is fully supported out of the box.

  Other databases will require:

  - Testing for compatibility with the implementation of Hibernate HQL queries and any Hibernate dialect issues
  - implementation of one stored procedure if instrument versioning will be used in LAVA
  - Utility stored procedures for developing instruments and LAVA Query objects will need to be implemented if they are to be used.
  - The LAVA Query tool will require that a number of stored procedures be implemented.

  Note that all of the above stored procedures can be ported from the existing MySQL versions.

- The Eclipse development platform could be replaced by another development platform, or even command line development

- The Apache Tomcat web application server, also known as a web container, can be replaced by any web container that implements the Java Servlet and JavaServer Pages (JSP) specifications, e.g. JBoss.

- The Ant build tool can be replaced by Maven or other build tools

MySQL

LAVA requires MySQL version 5.5 or later. If this is not already installed on your system, download and install MySQL and MySQL Workbench.

e.g. Mac OS X installation 10/3/12
MySQL Community Server 5.5.27 (x86, 64-bit) DMG Archive
Note: installation included instructions specific to Mac OS X to install the Startup Item and added icon to Mac OS X Settings to Stop/Start MySQL Server (and a flag to automatically start on startup).
Installed to:
/usr/local/mysql-5.5.27-osx10.6-x86_64
with the following symlink:
/usr/local/mysql

MySQL Workbench DMG Archive

Follow recommended procedure to set a password for the MySQL root user:
Note: without this can get in via "mysql -u root" without the –p for password prompt

Here are two alternate techniques to do this:

1)
```
./mysqladmin -u root password 'password'
```

2)
```
$ mysql -u root
mysql> use mysql;
mysql> update user set password=PASSWORD("NEWPASSWORD") where User='root';
mysql> flush privileges;
mysql> quit
```

Java

If Java is not installed already, install Java 1.6 (J2SE 6) or 1.7 (Java SE 7).

The development machine needs the JDK not just the JRE. While Eclipse has its own Java compiler, you still need the JDK in order to get the Java source code. And building LAVA applications uses the Java compiler installed on the machine, not the Eclipse Java compiler, and so the JDK is needed for this (this is done by configuring Eclipse to use the Ant build tool via Eclipse External Tools to compile the source code).

Note: Java JRE and JDK already installed on Mac OS X 10.7 (Java 1.6)

Eclipse

The Eclipse Development platform is an excellent platform for developing LAVA applications.

Download and install latest version of Eclipse.

Use the Eclipse IDE for Java EE Developers (rather than Eclispe IDE for Java Developers or Eclipse Classic) because the EE version is bundled with everything to start building web applications with support for Java Servlets and JSP.

e.g. 10/3/12 installation:
Eclipse Java EE IDE for Web Developers.
Version: Juno Release
Build id: 20120614-1722
Mac OS X: unzipped eclipse-jee-juno-macosx-cocoa-x86_64.tar.gz somewhere and then dragged the resulting Eclipse folder into Finder Applications

Make sure that Eclipse is set up to find the correct JRE on your system. Go to Eclipse Preferences, Java, Installed JREs. The location of the JRE to use is OS specific and not covered here.

Apache Ant

The Apache Ant build tool is configured as an External Tool in Eclipse to build and deploy LAVA applications, including hot deployment of JSP and i18n files.

Download latest version of Apache Ant from ant.apache.org

e.g. 10/3/12 installation
apache-ant-1.8.4-bin.tar.gz
cp to /usr/local and unzip

Configure in startup script, e.g. .bash_profile

```
export ANT_HOME=/usr/local/apache-ant-1.8.4
PATH=${PATH}:${ANT_HOME}/bin
export JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
PATH=${JAVA_HOME}/bin:${PATH}
```

Note: on Mac OS X determined JAVA_HOME from the Eclipse IDE looking at the Eclipse Preferences, Java, Installed JREs

Apache Tomcat Web Application Server

Apache Tomcat is an open source implementation of the Java Servlet and JavaServer Pages (JSP) technologies, per the Java Servlet and JavaServer Pages (JSP) specifications developed under the Java Community Process.

At UCSF LAVA is currently developed and used in production with version 6 of Apache Tomcat but you can try to use other versions (e.g. Tomcat 7) or other web application servers, as long as they comply with the Java Servlet Specification. The instructions which follow are for Tomcat 6.

On *nix systems, download Tomcat 6 from tomcat.apache.org and unzip in /usr/local (using sudo if necessary).

e.g. the following will be the resulting Tomcat installation directory:
/usr/local/apache-tomcat-6.0.35

On *nix systems, in order to run Tomcat as a user other than root, you should create a group, e.g. “tomcat”, and give that group ownership of the Tomcat installation and read/write/execute permissions. This will allow members of the group to stop and start Tomcat, and allow Tomcat to write to its directories for logging, etc.

e.g. change the owner to the group “tomcat” and give that group read/write/execute permissions, assuming install in /usr/local/apache-tomcat-6.0.35

```
sudo chgrp -R dev apache-tomcat-6.0.35
sudo chmod -R g+rwx apache-tomcat-6.0.35
```

Edit Tomcat configuration files as follows:

1) conf/web.xml: add trimSpaces true so that HTML source code generated from JavaServerPages will be much more readable:

```
<servlet>
<servlet-name>jsp</servlet-name>
<servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
<init-param>
<param-name>fork</param-name>
<param-value>false</param-value>
</init-param>
<init-param>
<param-name>xpoweredBy</param-name>
<param-value>false</param-value>
</init-param>
  →
<init-param>
<param-name>trimSpaces</param-name>
<param-value>true</param-value>
</init-param>

    <load-on-startup>3</load-on-startup>
</servlet>
```

2) conf/context.xml: follow instructions within to comment out Manager tag to disable persisting Tomcat sessions between restarts

3) conf/tomcat-users.xml: configure a role and user to authenticate the Tomcat Manager app by inserting the following lines (if desired, substitute your own username and password):

```
<role rolename="manager-gui"/>
<user username="tomcat" password="_PASSWORD_" roles="manager-gui"/>
```

note: the Tomcat Manager app can be a security risk so make sure that the password is very secure

Install a JDBC connector for MySQL in Tomcat.

The connector, called Connector/J, can be downloaded from mysql.com. Alternatively, for convenience, there is a copy of the file in the lava-core CVS project:

lava-core/tomcatlibs/mysql-connector-java-5.1.8-bin.jar

which you will obtain after you have configured Eclipse for LAVA CVS access in the next section (so you can could come back and do this step later).

Copy the Connector/J jar file to the Tomcat top-level lib directory, e.g.

```
cd ~/Downloads
tar xvfz mysql-connector-java-5.1.22.tar.gz
cp mysql-connector-java-5.1.22/mysql-connector-java-5.1.22-bin.jar  /usr/local/apache-tomcat-
6.0.35/lib
```

Install the Java Transaction API (JTA)

The following library is required by the HibernateTransactionManager (it is not clear why since not using distributed transactions, but LAVA will not run without it).This can be downloaded, using a search to find it. Alternatively, for convenience, there is a copy of the file in the lava-core CVS project:

lava-core/tomcatlibs/jta1-1.jar

which you will obtain after you have configured Eclipse for LAVA CVS access in the next section (so you can could come back and do this step later).

Copy the JTA jar file to the Tomcat top-level lib directory, e.g.

```
cp jta-1.1.jar /usr/local/apache-tomcat-6.0.35/lib
```

Configure Logging

The following steps configure Tomcat for Apache Commons Logging to work with Log4j. You can reference the following page online, but all the instructions are below:

tomcat.apache.org/tomcat-6.0-doc/logging.html

1) log4j.properties

Create a log4j.properties file with the following contents in the Tomcat /lib directory:

```
log4j.rootLogger=debug, R
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=${catalina.home}/logs/tomcat.log
log4j.appender.R.MaxFileSize=100MB
log4j.appender.R.MaxBackupIndex=10
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %d{MM/dd HH:mm:ss} %c - %m%n
log4j.logger.org.apache.catalina=DEBUG, R
```

2) jar files installation

Go to the Tomcat 6 Download page, click Browse, go to bin/extras and download:

- tomcat-juli.jar (This tomcat-juli.jar differs from the default one. It contains the full Apache Commons Logging implementation and thus is able to discover the presence of log4j and configure itself)
- tomcat-juli-adapters.jar

Copy these files per the following examples:

```
cp tomcat-juli-adapters.jar /usr/local/apache-tomcat-6.0.35/lib
cp tomcat-juli.jar /usr/local/apache-tomcat-6.0.35/bin  NOTE: that's bin not lib
```

3) Apache log4j (version 1)

Download Apache log4j from logging.apache.org (Log4j 1) and copy the jar file per the following example:

```
tar xvfz apache-log4j-1.2.16.tar.gz
cp apache-log4j-1.2.16/log4j-1.2.16.jar /usr/local/apache-tomcat-6.0.35/lib
```

4) conf/logging-properties

Delete conf/logging-properties to prevent java.util.logging from generating zero length files

Create Directory for Context Descriptor

A context descriptor is an XML file that contains Tomcat related configuration for a Context, i.e. the LAVA web application.

The technique used here is to put context descriptors for LAVA applications in the Catalina/localhost directory, i.e. each LAVA webapp has a context descriptor file in this directory.

Follow this example:

```
cd /usr/local/apache-tomcat-6.0.35/conf
mkdir Catalina
cd Catalina
mkdir localhost
```

NOTE: do not have to define CATALINA_HOME for the Apache Tomcat configuration. This is automatically setup in the Tomcat startup scripts.

Configuring Eclipse to Launch Tomcat

Eclipse can be configured so you can start and stop the Tomcat server.

The following instructions are based on the current version of Eclipse as of the time of this writing, i.e. Eclipse Juno. Menus and text may differ in different versions of Eclipse.

You can reference the following instructions online but all instructions are included below;

http://theopentutorials.com/tutorials/java-ee/how-to-configure-apache-tomcat-in-eclipse-ide/

1) In Java or Java EE Perspective, go to Servers view, right-click, New, Server

Recommend the default configuration:
Server's host name: localhost
Server name: Tomcat v6.0 Server at localhost
Server runtime environment: Apache Tomcat 6.0

2) Click Configure Runtime Environments…

Choose the Server just created and Edit, making sure the Tomcat installation directory is correct:
Name=Apache Tomcat v6.0
Tomcat installation directory=/usr/local/apache-tomcat-6.0.35 (or whatever version)
JRE=Workbench default JRE

Finish, OK, Finish to complete adding server (do not need to click Next on New Server wizard as there are not resources to be configured on the server).

Note: alternatively to 1) and 2), you can go to Eclipse, Preferences, Server, Runtime Environments, and adding a Runtime Environment (using the same settings as above) will also add the server to the Servers view, as long as you have checked "Create a new local server".

3) In the Servers view (by default, a tab where the Console view is), open the Tomcat server to open up the configuration, then:

- In the Server Locations section, make sure "Use Tomcat installation" is checked
- In the Timeouts section, make sure you specify enough time for application to start and stop (e.g. start with 180 and 60 seconds, respectively)

Save

- Modify your launch configuration by clicking on "Open launch configuration" link
    - Click on the Arguments tab
    - Modify the VM arguments section to look like the following:

NOTE: the following was already set up by virtue of having created the server so just have to add the memory params below:

```
-Dcatalina.base="/usr/local/apache-tomcat-6.0.35"
-Dcatalina.home="/usr/local/apache-tomcat-6.0.35"
-Dwtp.deploy="/usr/local/apache-tomcat-6.0.35/wtpwebapps"
-Djava.endorsed.dirs="/usr/local/apache-tomcat-6.0.35/endorsed"
-Xms512m
-Xmx1024m
-XX:PermSize=256m
-XX:MaxPermSize=256m
```

The above –X memory arguments to the Java Virtual Machine may need to be adjusted depending upon the machine configuration (memory) and size of the application. In particular, the PermSize arguments can be adjusted to combat the occurrence of the infamous PermGen error which crashes the Tomcat application server.

Start Server

It is easy to manage the server instance. In the “Servers” view (e.g. in the Java Perspective) there are icons to start the server/start the server in debug mode. If you have multiple servers configured you will need to highlight the correct server before starting. When a server has started you will see an icon to stop the server.

You can also right-click on the server in the Server view and start and stop it

Test your installation

After starting the Tomcat, open browser and type http://localhost:8080. You should see the Apache Tomcat home page as shown below.

You can also run the Tomcat manager web app, using the user and password configured in tomcat-users.xml above (e.g. user="tomcat", password="_PASSWORD_")

http://localhost:8080/manager/html

Note: the Tomcat Manager is also accessible from the Apache Tomcat home page

LAVA Source Code Repository

All of the source code and other web application elements of the lava-core, lava-crms, lava-crms-nacc and lava-app-demo Eclipse projects as well as all other UCSF LAVA web applications are stored in a CVS source code repository hosted at the UCSF Memory and Aging Center. While the source code is freely available (once the LAVA Software Transfer Agreement has been signed), for quality reasons, the source code in the LAVA CVS repository can only be modified by members of the UCSF LAVA development group. Those outside of this group are encouraged to make modifications such as new features, bug fixes and other code improvements. In this case, the code is submitted to the LAVA development group (generally via email) and the LAVA development group then makes the changes. Ideally, code changes should be submitted as Eclipse patch files, as these can be easily applied and minimize errors when merging code.

Those LAVA developers outside of the LAVA development group will have readonly access to the LAVA CVS repository so that they can obtain the latest at any time.

When customizing a LAVA application, developers should use the customization techniques supported by the LAVA software architecture. Refer to the "Customizing and Extending" section for details. If this is done properly, there should not be any changes within the lava-core, lava-crms and lava-crms-nacc projects as a result of the customization. This allows developers to keep their lava-core, lava-crms, and lava-crms-nacc source code in sync with the CVS repository so that all LAVA developers are using the identical source code in those projects. Any bug fixes, features, or other improvements to those projects will then automatically propagate to any LAVA web application that uses them, simply by doing a CVS update.

If source code changes to lava-core, lava-crms or lava-crms-nacc also involve database changes, i.e. database schema or metadata changes, then these should be submitted to the LAVA development group as SQL scripts. The script will be committed to the CVS repository under the "database" directory of the project, possibly integrated with a larger update script, with accompanying documentation guiding LAVA developers who wish to upgrade.

Configuring LAVA CVS Repository in Eclipse

Setup connection to the LAVA CVS source code repository as follows:

1) Open Eclipse Perspective - CVS Repository Exploring

2) Add CVS Repository with the following parameters:

```
Host=mac139.ucsf.edu
Repository path=/usr/local/cvsroot
User=lava-readonly
Password=l@v@
Connection Type=pserver
Use Default Port
```

Access to this repository is firewall protected, so before you will be able to connect to it, you will have to provide the IP address(es) of your development computer(s) and request that the firewall be configured to allow access. Contact the LAVA source code repository adminstrator for access to the repository.

Obtaining LAVA source code

See the “Development Environment Installation” section for instructions on configuring the LAVA CVS repository in Eclipse.

Create the Eclipse projects from the CVS repository

1) Open Eclipse Perspective - CVS Repository Exploring

2) Choose cvs module: HEAD/lava-core, right-click "Check Out As.."
   select "Check out as a project in the workspace"
   Project Name:lava-core
   Checkout subfolders: yes
   Add project to a working set: no
   Finish

   Repeat Step 2) for:
   HEAD/lava-crms
   HEAD/lava-crms-nacc (optional, for UDS instruments)
   HEAD/lava-app-demo (optional, to build the LAVA Demo app)

note: for your own application you will create your own lava-app-MYAPP Eclipse project, but for reference purposes you can retrieve lava-app-demo and build/deploy the LAVA Demo application. Refer to the “Creating a New App from the LAVA Demo App” section of this document for creating your own lava-app-MYAPP project using lava-app-demo as a starting point.

You now should have an Eclipse project for each CVS module you retrieved above.

For the LAVA Demo web application there will be four Eclipse projects; lava-app-demo which in turn uses code and resources from lava-core, lava-crms and lava-crms-nacc.

If the Eclipse Package Explorer is treating every top-level directory as a Java package, then you must exclude from the Build Path every top-level directory which is identified as a Java package (with the package icon) that is not in reality a Java package (only the “src” directory is a Java package directory):

Right-click on a top-level directory, choose Build Path / Remove from Build Path. Repeat for every directory except for “src”.

Note: there are other ways to do this such as configuring the Java Build Path in Project Properties

LAVA Relational Database

LAVA supports any relational database for which the Hibernate Object Relational Mapping layer offers support. Hibernate provides a layer of abstraction that allows switching to another underlying database technology.

Existing LAVA applications use the MySQL database. If you are going to build a new LAVA application that uses a different underlying database then you will need to convert anything that is proprietary to MySQL in the application template that you created from the LAVA Demo application. This includes the following database triggers and stored procedures:

- for lava-crms-nacc applications, database triggers are used to change the version of an instrument from one version to another
- for instrument creation there are a series of utility stored procedures which automate some of the process of creating a new instrument in LAVA
- for the LAVA Query tool, there is a stored procedure “layer” which facilitate the database queries made by the query tool

If using MySQL you should be using MySQL 5.5 or later because of support for the sha2 function which is used by the LAVA query tool to authenticate LAVA “local” account passwords such that a LAVA user can use their LAVA account for LAVA Query as well.

Schema

The easiest way to get up and running with a new application is to run the latest “complete” script in the lava-app-demo project, followed by running any update scripts created since the latest completed script was released. A "complete" script contains everything in the database needed to run a LAVA web application, i.e. all database tables, views, stored procedures and metadata.

First you must create your database user and database:

In MySQL create a database called "APP_DB" and "grant all on APP.*" to your MySQL user,

e.g. assuming username=DB_USER, if user does not exist yet:
```
mysql>create user DB_USER identified by 'PASSWORD';
mysql>create user 'DB_USER'@'localhost' identified by 'PASSWORD';
```

Create the database and grant DB_USER all rights in that database.
```
mysql>create database APP_DB;
mysql>grant all on APP_DB.* to DB_USER, 'DB_USER'@'localhost';
```
There are two variations of the lava-app-demo complete scripts, one with sample UCSF patient data (with de-identified patient names) including visit and assessment data, and another with no patient data, i.e. a “blank” database. Both complete scripts contain the schema and metadata to support the lava-crms-nacc project, i.e. the NACC UDS and FTLD instruments. If you would like a complete script for “blank” database without these, contact the LAVA development group.

Running Complete Database Script

Run the latest version of the complete script, e.g. to load the complete script without any data, using the latest version at the time of this writing:

(the following is all on one line)

```
mysql -u DB_USER -p APP_DB <
~/lava-app-demo/database/local/demo/versions/V3.1/V3.1.0/complete/lava-demo-complete-
uds-dump-3.1.0.sql
```

You should also run any update scripts created since the latest complete script, e.g. in the example above, since the V3.1.0 release. The section "Creating a New App from the LAVA Demo App" documents the practical application of creating a new LAVA application and running the complete and update database scripts.

Alternatively, the database schema for an app can be created by running model and data scripts for each Eclipse project. The model script creates the database tables, and the data script inserts metadata, such as list data and entity property metadata, which describes how a property is displayed in the view (i.e. in a form on a web page).

*** This is outlined below for illustrative purposes only ***. Generally speaking you will always want to run a complete script for a new app that goes with the latest version of the source code The complete and update scripts are subject to greater quality control than the model and data scripts so are considered to be the definitive schema with data to create for a new app.

For a given project, these model and data scripts are stored as follows:

lava-core:
```
~/lava-core/database/core/versions/V../model/*
~/lava-core/database/core/versions/V../data/*
```

lava-crms:
```
~/lava-crms/database/crms/versions/V../model/*
~/lava-crms/database/crms/versions/V../data/*
```

lava-crms-nacc:
```
~/lava-crms-nacc/database/crms-nacc/versions/V../model/*
~/lava-crms-nacc/database/crms-nacc/versions/V../data/*
```

APP:
```
~/lava-app-APP/database/local/APP/versions/V../model/*
~/lava-app-APP/database/local/APP/versions/V../data/*
```

The latest version script from the model and the data directories should be run, using MySQL command line, e.g.

```
mysql -u DB_USER -p APP_DB < ~/lava-core/database/core/model/lava-core-3.0.2.sql
mysql -u DB_USER -p APP_DB < ~/lava-core/database/core/data/lava-core-data-3.0.2.sql
```

In theory, you would repeat the above for each project, if you were not creating the database schema via the recommended procedure of running the complete script.

Generating Complete Database Script

Complete database scripts are generally created and used when an application is ready for an initial release. A complete database script is the mechanism whereby the database is transferred from development to a staging or production database.

The complete database script contains the complete schema for the application, and accompanying configuration data, metadata, and other pre-populated data, such as authorization data. The script may also contain actual data that has been migrated from another data source, such as CSV files exported from a legacy system.

For MySQL the complete script is generated using the mysqldump command. The recommended syntax for creating a complete script is:

```
mysqldump -R -c --comments=FALSE  --tz-utc=FALSE  -u DB_USER -p APP_DB > lava-app-demo-dump.sql
```

In order for the complete script to be executable by a non-root user in the target database, you may need to edit the mysqldump output file and remove all occurences of DEFINER=

e.g. if your mysqldump script generated DEFINER clauses, e.g.
DEFINER=`demo`@`localhost`
use an editor to replace all such clauses with blank

The complete script can now be executed on the target database, e.g.

```
mysql –u DB_USER –p APP_DB < lava-app-demo-dump.sql
```

Metadata / List Data

Metadata scripts come into play during development when the app is modified with customizations and extensions, and they support both multi-programmer development (where each programmer has their own local development database) and the release process in transferring metadata from a development database to the production database. The process for working with the metadata scripts is covered in the “Building LAVA Applications” section and metadata is used in the “Extending and Customizing” section.

There are five metadata tables:
viewproperty
hibernateproperty
list
listvalues
datadictionary

A description of the purpose of each table follows.

viewproperty
Used at runtime to determine how a property will display in a view. Used during development for code generation (see Creating Instruments section and util_GetCreateFieldTags and util_GetResultFields)

hibernateproperty
Not used at runtime. Used during development to for code generation ((see Creating Instruments section and util_GetJavaModelProperties and util_HibernateMapping)

list and listvalues
Used at runtime to build all of the lists used for populating user interface controls. list contains the list names, and listvalues contains the items for every list.

note: LAVA also supports dynamic lists, which are created via queries rather from data in the listvalues table

datadictionary
Not used at runtime. This serves as a datadictionary for entity properties.

The metadata files have both “instance” and “scope” columns. They are part of the mechanism used by an application to extend and customize LAVA. See the “Customizing and Extending” section for practical application of “instance” and “scope” in the metadata tables. See the “Development Framework” section for a more in depth description.

“instance” and “scope” are also used to facilitate the following functions:

1) For multi-programmer development as a way to sync metadata changes between developers

2) For application releases to transfer this metadata from the development database to the production database.

lava-core project metadata has ‘lava’ instance, ‘core’ scope lava-crms project metadata has ‘lava’ instance ‘crms’ scope

lava-crms-nacc project metadata has ‘lava’ instance ‘crms-nacc’ scope

And an application will have its own instance and scope, e.g.
lava-app-xyx metadata has ‘xyz’ instance and ‘app-xyz’ scope

note: the application scope can be named whatever the developer wants, but must be used consistently as the value for scope throughout the app

The application instance is the same as the context path of the application in Java Servlet Specification terms, e.g. “xyz”. There could be more than one distinct configuration of an application, in which case there will be additional instances, e.g. ‘xyz-training’

Building LAVA Applications

This section discusses everything pertaining to building your LAVA application. References are made to software installed in the “Development Environment Installation” section, such as the Eclipse development platform.

Eclipse LAVA Java configuration

Configure Java in Eclipse so that Eclipse's built-in Java compiler will be able to flag compiler errors.

The LAVA specific part of this configuration involves configuring the Eclipse “Java Build Path” (right-click project in Eclipse and choose Properties to see Java Bild Path). Each project in Eclipse has a .classpath and a .project file. The Eclipse .classpath file contains the Java classpath used by its compiler. It also contains references to other Eclipse project upon which the project depends for its compilation. Changes to the Java Build Path are reflected in .classpath and vice versa. Note that only those .jar files that are needed for compilation need to be in the Java Build Path, so .jar files in WEB-INF/lib that are only used at run-time do not need to be in the Java Build Path (.classpath).

All the jar files used in a LAVA application are stored in the lava-core project in the WEB-INF/lib folder. This is true even if a jar library is used by Java source code in a project other than the lava-core project. This is done to simplify the setup by having all jar files live in one place. After all, when the web application is deployed, there is only one WEB-INF/lib folder, so the jar files from all projects would get combined in a single folder eventually.

While the Eclipse Java Build Path is used by the Eclipse Java compiler, when a project is built (i.e. compiled), deployed and packaged using the Ant build tool and the project’s build.xml, the Java compiler installed on your system is used (later in this document you will set up Ant build targets in Eclipse via Eclipse External Tools). The build.xml sets up its own classpath which is basically all of the .jar files in lava-core WEB-INF/lib, as well as any external .jar files (i.e.g. servlet-api.jar).

As far as the Java run-time classpath goes, the Java Servlet Specification defines the directory structure of a web application, specifying where all .class and .jar files should live, so it knows where everything is at run-time. If there are any external libraries that are needed to run the application server (i.e. Tomcat) they should be added to the Launch Configuration of the server as configured in Eclipse (see the “Development Environment Installation” section for info about the Launch Configuration). By default the Java JRE and related libraries will be referenced in the Launch Configuration. The servlet-api.jar need not be since that is in the application server library itself (as is the database JDBC driver for the database connection).

NOTE: The Eclipse project files .project and .classpath should NOT be in CVS but they currently are. They should not be in CVS because these are specific to the Eclipse installation and development setup and also because somebody should be able to develop LAVA applications with any editor/developer IDE. However, for the sake of efficiency in getting set up for LAVA development, these files are in CVS and they are set up such that they generally apply for everyone. Because of this, you will not need to do Step 1) or 2) below from scratch; you can start with .classpath and .project from CVS and make any necessary adjustments.

For your lava-app-MYAPP project, if it was created following the instructions in the "Creating App from Demo App" section of this document, then its .project file will be set correctly, and its .classpath file will be the .classpath file from lava-app-demo.

Note that with Step 2), the .classpath file from CVS specifies all of the .jar files used by LAVA, whereas you really only need those which are required for compiling; the rest of them are only required at runtime.

1) Make sure each Eclipse project is a "javanature" project. edit .project file and it should have this in it:

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
 <name>lava-app-APP</name>
 <comment></comment>
 <projects>
 </projects>
 <buildSpec>
  <buildCommand>
   <name>org.eclipse.jdt.core.javabuilder</name>
   <arguments></arguments>
  </buildCommand>
 </buildSpec>
 <natures>
  <nature>org.eclipse.jdt.core.javanature</nature>
 </natures>
</projectDescription>
```

This also includes defining the Java Builder (which can also be turned on in the Eclipse Project Properties under Builders). The Java Builder builds programs incrementally as files are saved, using Eclipse’s built-in Java compiler, visually marking problems as warnings or errors. We are not using Eclipse to build our LAVA applications; rather we are using Eclipse’s External Tools to configure Ant Builds to build our LAVA application with the Java compiler installed on the development machine. Nevertheless, it is quite useful to have the Eclipse Java Builder configured to notify about compiler warnings and errors during development.

2) Setup Project/Propeties/Java Build Path so that Eclipse can compile your source.

The .classpath file from lava-app-demo can be used as a starting place for your lava-app-MYAPP project.

Source tab
lava-app-APP/src
IMPORTANT: if your project does not have a "src" folder off of the root, you should create it

Projects tab
lava-core
lava-crms
lava-crms-nacc

Libraries
note: if got .classpath file from CVS, build path already set up but may need to be adjusted

Add JARs, browse to lava-core project's WEB-INF/lib
spring.jar
spring-webflow-1.0.4.jar
spring-binding-1.0.4.jar
hibernate3.jar
jasperreports-3.1.2.jar
commons-*.jar
...
etc..

NOTE: as the jar files used is dynamic over time, consult the latest .classpath in CVS for lava-core, lava-crms, lava-crms-nacc and lava-app-demo for an up-to-date list.

Add External JARs

If your lava-app-MYAPP project has Java source files that reference the Java Servlet API, then you will need to add that .jar file to the Java Build Path:

For Apache Tomcat the servlet-api.jar file in the lib directory, e.g.

/usr/local/apache-tomcat-6.0.20/lib

There is also an Apache Tomcat servlet-api.jar file in the LAVA source code repository for convenience if needed:

lava-core/tomcatlibs/servlet-api.jar

note: generally speaking, you should use the servlet-api.jar file specific to your Web Container, e.g. Apache Tomcat, which is why this is an external jar.

Order and Export tab

nothing

NOTE: in Project Properties, Java Build Path, Source tab, the Default Output Folder is set to bin. This is irrelevant because our Ant build file explicitly outputs compiled .class files to the WEB-INF/classes directory, in accordance with the Java Servlet Specification.

Eclipse LAVA Projects

Upon building the web application, all of the files required by Tomcat for deployment are combined from lava-app-APP, lava-crms, lava-crms-nacc (optionally) and lava-core into a single "deploy" directory. This is all managed by the Ant build.xml file, which is in the lava-app-APP project root, along with the build.properties file which contains user specific settings.

Following is a description of each Eclipse LAVA project. These are also covered in the “Development Framework” section.

lava-core
Core plumbing and infrastructure support for a web application framework (utilizing the Spring framework), authentication, authorization, auditing, HTTP session management, webflow (which controls the web page flows used for viewing, editing, deleting, lists, etc.), reporting, and generic entity CRUD support (utilizing the Hibernate Object Relational Mapping layer between your Java classes and MySQL)

lava-crms (crms=Clinical Research Management System)
Expands on lava-core and adds support for Projects (studies), Patient, Enrollment (in a project), Visit, and Assessments (individual assessments are called instruments).

lava-crms-nacc
Contains the implementation of all of the NACC UDS and FTLD instruments

lava-app-demo
Customizes the application in terms of what tabs should be shown, configures the Java JDNI data source for connecting to MySQL, configures what kind of authentication can be used for logins, etc. This is also where customizations to the standard lava-crms can be done, e.g. if you want to add or modify a field to the Patient or Visit. There is also customization you will see in the demo app for configuration of UDS visits.

The Ant build.xml file is the key to how all of these different Eclipse projects come together to create a single deploy directory for the webapp. Our naming convention is such that lava-app-APPNAME is used for application level projects, i.e. projects that when combined with scope level projects(s) (e.g. lava-core and lava-crms) result in a deployable web application. For this reason, the lava-app-APP project is the project from which builds will be run. You will notice that the lower level projects, lava-crms and lava-core, do not have a build.xml file (and therefore do not need a build.properties file) because they are are not complete web applications by themselves. It only makes sense to compile and deploy their files within the context of a web application project, such as lava-app-APP.

Our naming convention for non-application projects is lava-SCOPE, and currently there are two scopes, "core" and "crms", so those are the lava-core and lava-crms projects.(crms = Clinical Research Management System).

The other Eclipse project, lava-crms-nacc, is a bundle of files that belongs in the "crms" scope but which has been separated out because it contains the implementations for all of the NACC UDS and FTLD instrument forms, and some crms applications will want these, while others will not. Basically the lava-crms-nacc project represents a technique to create a reusable bundle of resources for a given scope, which applications then may or may not include. The naming convention is lava-SCOPE-BUNDLE.

When developing, the Ant build.xml file in lava-app-demo (along with build.properties) contains tasks that combine the source from the above four projects into a single deploy directory, which is structured (in terms of file and directory naming and locations) as designated by the Java Servlet Specification, so that a servlet container such as Apache-Tomcat can load the application.

If you study the build.xml for lava-app-APP, you will see how it brings in these other projects, e.g. look at the "compile" target and you can see that it compiles files from all of the projects that it needs in order to run. These files are compiled such that the resulting .class files are created in the deploy directory, defined by instance-deploy.path in build.properties. Besides Java .class files, all other files that are needed by lava-app-APP are copied to this "deploy" directory via the Ant "copy" target.

note: Eclipse tip. In Eclipse the default editor for an xml file is not a text editor, so unless you prefer the XML editor do a right-click Open With on xml files and choosing Text Editor (you only have to do this once for a file and it will remember)

Build Targets

For the most part, you will be running the following Ant targets:

compile = compile all of the files in your project as well as the files in lower level projects and locate the .class files in the deploy directory.

copy = copy all non .class files from your project as well as the files in lower level projects needed for webapp deployment into the deploy directory. if the only changes made are to .jsp files, these changes will take effect simply by executing the "copy" target.

merge-i18n-files = copy all non .class files from your project and merge the i18n files (e.g. messages.properties) from all Eclipse projects into a single file. In conjunction with configuring your application such that these message files are reloadable (by setting cacheSeconds to 0 in core-metadata.xml, this target facilitates an instantaneous way for message property changes to take effect.

deploy-reload = compile and copy as above, and reload the webapp

rebuild-deploy = delete the deploy directory, compile and copy as above. this target will be needed in cases where the deploy directory is out of sync with the source for some reason, e.g. this can happen when changes made to a source file have been copied to deploy, and then the changes are rolled back (using the Eclipse Replace with/Latest from HEAD feature to go back to the latest version in CVS). In this case, the source file will acquire an earlier timestamp than the version in the deploy directory, so it will not be moved over by the Ant copy target. So the deploy directory needs to be cleaned (deleted).

Dissecting the target names, you can think of things like this:

deploy = copy all files necessary for running the webapp to a deploy directory

reload = the server reloads the web application in the deploy directory into memory

rebuild = the deploy directory is deleted

note: the use of "deploy" is not exactly the way deploy is defined in Tomcat; deploy is a combination of the deploy and reload targets, where the end result is that the web application is installed in a running Tomcat server, i.e. it is running in memory.

build.properties

build.properties belongs in the root directory of your LAVA application, in the same directory as build.xml

Because build.properties is specific to the development environment, it should not be kept in CVS. You can control this by doing Team/Add to .cvsignore for this file, or just make sure you never commit it.

The contents of build.properties is:

```
instanceName=_APP_CONTEXT_PATH_
workspace.path=_PATH_TO_ECLIPSE_WORKSPACE_DIR_
deploy.path=_PATH_TO_PARENT_DIR_OF_APP_DEPLOY_DIR_
warName=_NAME_OF_PACKAGED_WAR_FILE_
j2ee14lib.path=_PATH_TO_JAVA_SERVLET_SPEC_API_
```

e.g.

```
instanceName=demo
workspace.path=/Users/ctoohey/Documents/dev/workspace
deploy.path=/Users/ctoohey/Documents/dev/deploy
j2ee14lib.home=/usr/local/apache-tomcat-6.0.20/lib
warName=demo
```

In more detail:

instanceName should be set to your application context path, e.g.

```
instanceName=APP
```

deploy.path and instanceName are combined as follows to specify the location of your deploy directory, i.e. the directory to which the application files from all Eclipse projects (e.g. lava-app-xyz, lava-core, lava-crms, lava-crms-nacc) are copied in a structure that is in accordance with the Java Servlet Specification and that will be deployed by Apache Tomact:

```
${deploy.path}/${instanceName}.war
```

note: the requirement that the directory end in .war is a holdover from when JBoss was used

The deploy.path should be a separate directory from the Eclipse "workspace" directory, e.g. it could be a parallel directory called "webapps". Whatever is specified for deploy.path and instanceName here is what should be reflected in docBase described in the Tomcat Context Descriptor file below.

e.g.

```
deploy.path=/home/username/project/deploy
warName=APP
```

then deploy directory will be:

```
/home/username/project/deploy/APP.war
```

i.e. the deploy.path and instanceName combined (with .war extension) must be equivalent to the value of docBase in the Tomcat application context file for the app (set up later in this document)

workspace.path should be set to your Eclipse workspace area, i.e. the folder above your Eclipse project folders, e.g.
workspace.path=/home/username/workspace

j2ee14lib.home should be set to the location of the Java Servlet Specification API library, i.e. servlet-api.jar
e.g.
j2ee14lib.home=/usr/local/apache-tomcat-6.0.20/lib

warName is the name of the packaged WAR file created from the webapp per the Java Servlet Specification of the layout of a WAR file. This can include a path, otherwise it will be created in the directory where Ant is run. The build.xml package-war target operates on the deploy directory, where the files from all projects making up the app have been combined, to create a WAR file, e.g. that could then be moved to a production server.

notes:

because each of the .properties files (*mvc.properties, *messages.properties, etc.) are spread across different projects, the build.xml file also contains a target, “merge-i18n-files”, which concatenates the .properties files from different projects into a single file located in the deploy directory.

Eclipse Build Configurations

To build LAVA we use Ant and run Ant using the External Tools feature of Eclipse.

General Instructions to create a run configuration
1) Go to Run/External Tools/External Tools Configurations...
2) In the External Tools Configurations dialog, right-click Ant Build, New.
3) Create "compile" run configuration:
  Name=compile or _APP_-compile
  Buildfile - Browse Workspace, select build.xml (in project root directory)
  Targets tab - unclick usage and click the "compile" target
  Apply

4) Create "copy", "merge-i18n", "deploy-reload", "rebuild-deploy" External Tools run configurations using the same steps as above, choosing the correct target

note: the properties required by the build.xml file for each of the targets above are supplied from the build.properties file which is in the root of your lava-app-APP project. It does not exist in CVS because it is developer specific.

To run an External Tools Run Configuration
1) Go to Run/External Tools/External Tools Configurations
2) Select the desired configuration and click the Run button

Once a configuration is run it is added to the External Tools menu and to the Run toolbar button for quick access.

e.g. to build and deploy the LAVA demo app, assuming run configurations were created for the lava-app-demo Eclipse project, click the Run Tools icon (not the Run Tomcat server icon) at the top of Eclipse (in the second group of icons) and choose "demo-deploy-reload" which will then run Ant (which is the external tool) using the "deploy-reload" target of the lava-app-demo build.xml file. Any modified files will be copies to the deploy directory, any modified Java classes will be compiled to the deploy directory, and Tomcat will reload the web application

Tomcat Application Context Descriptor

A Context is what Tomcat calls a web application. In order to configure a Context within Tomcat a Context Descriptor is required. A Context Descriptor is simply an XML file that contains Tomcat related configuration for a Context, e.g naming resources for database connections.

There is a sample Context Descriptor file in the lava-app-demo project.
See database/local/demo/versions/V3.1/V3.1.0/config/demo.xml

Take note of the use of the docBase attribute to point Apache Tomcat to your development directory, and that this is not used when deploying a WAR file (see the "Production Release" section below for more information about the WAR file).

The Context Descriptor file looks like this:

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="/lava/dev/deploy/demo.war/" reloadable="true">
   <WatchedResource>WEB-INF/web.xml</WatchedResource>
   <Resource name="jdbc/demo" auth="Container" type="javax.sql.DataSource"
       maxActive="50" initialSize="10" maxIdle="10" minIdle="5" maxWait="10000"
       testWhileIdle="true" timeBetweenEvictionRunsMillis="60000"
       minEvictableIdleTimeMillis="90000"
       validationQuery="SELECT 1"
      username="demo" password="PASSWORD" driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost:3306/demo?autoReconnect=true"/>
</Context>
```

Use this descriptor file as a starting point, modifying it accordingly for your LAVA app.

Place your file in the conf/Catalina/localhost directory. If the "Catalina/localhost" folders do not exist, create these folders in your operating system under the Apache Tomcat conf directory

For example, the context Descriptor file would be placed under the following folder:
apache-tomcat-6.0.20/conf/Catalina/localhost

Development Cycle

Once the Apache Tomcat server is started and your app has a Context Descriptor file, you can now run your application.

Building Your App
To run one of the build configurations created above, the first time go to External Tools Configurations (either via the Run menu, or the Run Tool icon) and select the build configuration to run and click Run. Thereafter, this build configuration will be available as a shortcut off of the Run Tool icon at the top of Eclipse (in both the Java and Debug perspectives).

compile target
To compile the files in your app, run the "compile" build target. It will only compile Java files whose .java source file date is later than its corresponding .class file date.

rebuild-deploy target
If this is the first time building your app, or, if you have deleted source files that should be removed from the deploy directory, run the “rebuild-deploy" target. After running the "rebuild-deploy" target for the first time, the "deploy-reload" target needs to be run.

deploy-reload target
To load or run your app, make sure Apache Tomcat is running and your Context Descriptor xml file is in the Apache Tomcat "conf/Catalina/localhost" directory, and run the "deploy-reload" target. If your application successfully loads, you should be able to access your application from a web browser, e.g.
http://localhost:8080/_APP_

note: Apache Tomcat is pre-configured to serve http on port 8080

Assuming your application has already been deployed, the act of starting the Apache Tomcat server will also load your web app.

If your application does not run, you can use the Tomcat Manager app to see if it is loaded yet or not.
http://localhost:8080/manager/html
The login/password for your Apache Tomcat Manager are specific to your Apache Tomcat installation.

copy target
If you have only made changes to JSP-based files, including WEB-INF/jsp, WEB-INF/decorator/../*.jsp, and WEB-INF/tags, this type of file is hot deployed by Apache Tomcat. All you have to do is run the "copy" target. Any modified file(s) are copied to the deploy directory and your changes go into effect immediately.

merge-i18n-files target
If you have only made changes to .properties files in the WEB-INF/i18n folder and any JSP-based files, there is a development mode setting which can be used to hot deploy these files.

In lava-core, modify
lava-core/WEB-NF/context/core/core-metadata.xml
in the "propertiesMessageSource" bean change the "cacheSeconds" attribute value from -1 to 0. This will result in checking all of the .properties files every time a message key needs to be looked up. However, do NOT do this in a production environment as performance will suffer; set this back to the value -1 in your production deployment.

Upon modification of this setting, the web application must be reloaded for the change to go into effect.

After configuring the .properties files to hot deploy, whenever changes are made to a .properties file, run the "merge-i18n-files" target. Any modified file(s) are copied to the deploy directory and changes go into effect immediately.

Metadata

In development mode, where your app schema already exists, your most frequent database related changes will be to the metadata.

For practical applications of using metadata see the “Extending and Customizing” section.

Your metadata changes will be made directly to your development database, so they will be loaded when you load your app (or just reload the metadata itself) and nothing else need be done during this development cycle.

However, you will want to use metadata scripts in the following situations:

1) If another developer has updated any of the metadata that your application uses, then the way you would get the updates is by getting the latest version of each applicable metadata script from the repository and executing it.

2) If you make metadata updates that another developer will need to apply, then you will need to generate the metadata script for each scope where you made updates, and commit them to the repository.

3) If you are ready to deploy your application on a staging, production or other database, you need to generate the metadata script for each scope where you made updates so that you can then run each script on the target database (and commit each to the repository).

Note: This assumes that your target database already has the complete schema and you are making updates. If you are deploying to a database for the first time, then you will need to create a complete database script (e.g. using mysqldump) to be executed on the target database.

See the "LAVA Relational Database" section for an overview of the LAVA metadata scripts.

Generating Metadata Scripts

The examples shown here are for the lava-app-demo application but would apply to any application substituting the application name for "demo".

Each application should have the following metadata extract files:
lava-app-demo/database/local/demo/development/data/extract.script
lava-app-demo/database/local/demo/development/data/extract-app-demo-data.sql

NOTE: if you have just created your app from the LAVA Demo app, if you have not done so already, you will need to edit the database/local/_CONTEXT_PATH_/development/data/extract.script to replace the name of the database and database user with those for your app
To generate the metadata script for application scope, execute the MySQL command in the extract.script, which uses extract-app-demo-data.sql to generate:
lava-app-demo/database/local/demo/development/data/lava-app-demo-data.sql

This script can then be committed to the repository and run by another developer or on another database.

Each LAVA project has these scripts in its development/data directory, e.g.
lava-core/database/core/development/data/extract.script
lava-core/database/core/development/data/extract-core-data.sql

lava-crms/database/crms/development/data/extract.script
lava-crms/database/crms/development/data/extract-core-data.sql

If you have made changes to metadata in the scope of a project you should generate the metadata script and commit it to the source code repository.

IMPORTANT: before making changes to metadata in scopes that are used by other developers, make sure you obtain the latest version of the metadata from each scope and execute it on your development database, so that you do not overwrite changes made by other developers

Executing Metadata Scripts

Each metadata script is a SQL script which essentially deletes metadata for a particular instance and/or scope and then re-inserts the metadata for the instance/scope.
e.g. assuming you are using the metadata scripts in the “development/data” in the “database” folder, and your app used lava-core and lava-crms, you would open MySQL WorkBench (or use the mysql command line utility) to run the following scripts:

lava-core/database/core/development/data/lava-core-data.sql
lava-crms/database/crms/development/data/lava-crms-data.sql
lava-app-demo/database/local/demo/development/data/lava-app-demo-data.sql

note: the desired metadata script may be under “versions” instead of “development” and the actual script name may differ depending on the generation of the metadata script

Data Reload Shortcuts

The following URLs can be used during development to reload various kinds of data without having to reload the web application, which would normally be required for updates to these kinds of data to take effect.

The examples shown here are for the lava-app-demo application but would apply to any application substituting the application name for “demo”.

To reload all metadata request the following URL:

.../demo/admin/reload/metadata.lava

To reload all authorization data request the following URL:

.../demo/admin/reload/auth.lava

note: authorization permssions are cached for performance reasons so changes in permissions require reloading the web application, or the above shortcut

To reload all projects request the following URL:

.../demo/admin/reload/projects.lava

note: this is only applicable to applications that use lava-crms

Authentication

Users

Lava users have a username, a login, and an effective date. The authorization mechanism does not provide authentication services. This is handled in LAVA by the Acegi security framework that can be configured to authenticate user credentials with all major account management systems (such as Microsoft Active Directory, LDAP, OpenID). The login property of the user account simply needs to match the login supplied to the authetication service, and of course, all of this can be customized to meet specific requirements by subclassing the authentication code in LAVA for your applications.

Assuming you created your app from the LAVA Demo app using the instructions in the section "Create a New App from the LAVA Demo App", the default configuration comes configured with two users, configured via XML.

If your application instance is “xyz” the "admin" and "demo" users are configured in:
WEB-INF/context/local/xyz/xyz-security.xml

Each of these users has a record in the authuser database table. If you decide to modify or delete one of these users, you should make the corresponding changes in your application under the Admin module, Authorization section. If you change the login name (“demo” or “admin”) then the corresponding change must be made in the Admin module.

NOTE: **** you will not be able to do this if you have deleted the pre-configured "admin" user, so if you want to delete "admin", you should create another user with Admin privileges in the LAVA Admin module/Authorization section first. The Admin module tab only displays in LAVA if a user with Admin privileges is logged in **** (the pre-configured “admin” user password is whatever is configured in the xyz-security.xml file)

LAVA user accounts can be configured for one three types of authentication:

1) LOCAL accounts store the user login and password in the database where the password is encrypted
2) UCSF AD accounts authenticate using an account in the UCSF Active Directory
3) XML CONFIG accounts are configured in the security XML file that is used above for the “demo” and “admin” users

Regardless of the type of authentication, each user account must have a record in the LAVA authorization database tables, and these are created and modified in the Admin module, Authorization section.

You will typically create accounts with authentication type LOCAL for your users, as they can then change their own password in the Home module, User Info section of your app.

Groups

Groups are simply collections of users. Groups and user to group associations can be configured through the LAVA user interface

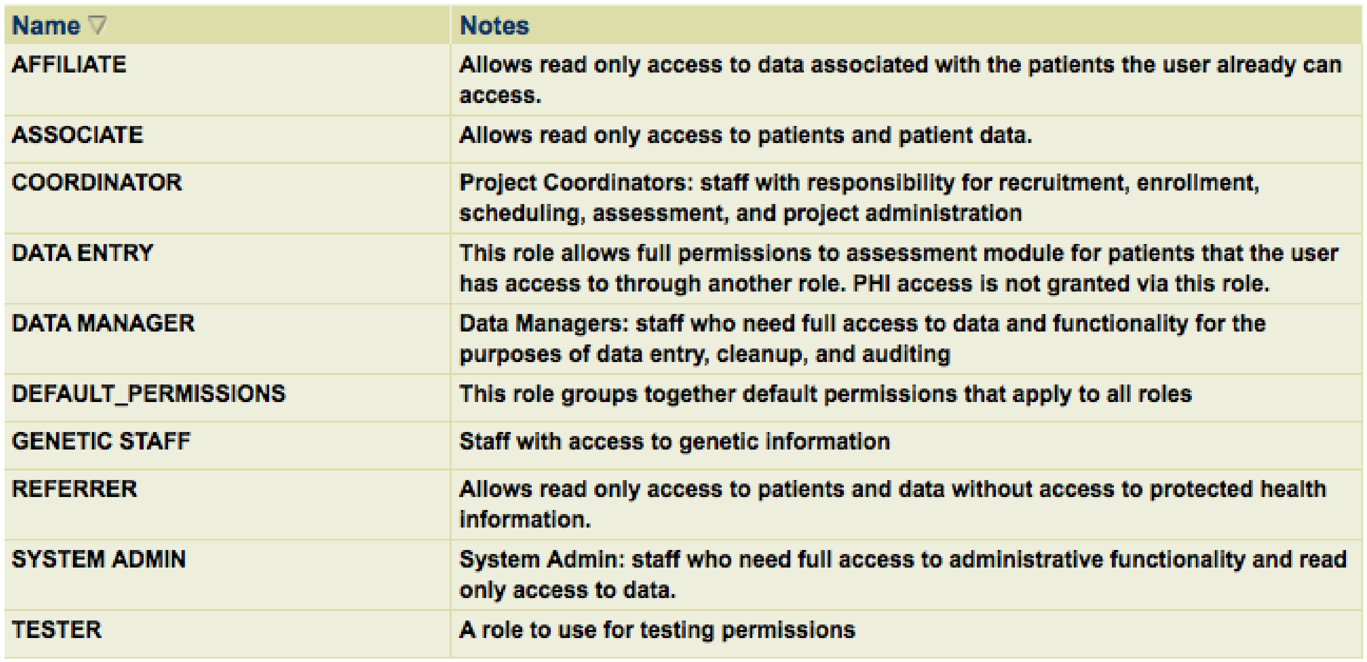
Authorization

In addition to authentication, the Admin module/Authorization section is used to manage all authorization, which can be configured for each user per LAVA project (not to be confused with Eclipse LAVA projects) and across all LAVA actions at the level of View/Add/Edit/Delete. LAVA Projects are part of the lava-crms project, where a project represents a research project or study. See the "LAVA Framework" section for more information about project context and action definitions.

Configuring authentication in LAVA consists of defining roles, assigning permissions to role, creating user accounts, creating groups, and assigning groups to roles. This section provides an overview of each of these activities.

Roles

Roles in LAVA are essentially a set of permissions to execute actions. Like any standard role-based security mechanism, determining the roles that you will need and the permissions that they should have is a critical part of effectively securing your applications. Examples of roles that we have found useful in securing CRMS applications are shown in the screen capture below:



Name ▽	Notes
AFFILIATE	Allows read only access to data associated with the patients the user already can access.
ASSOCIATE	Allows read only access to patients and patient data.
COORDINATOR	Project Coordinators: staff with responsibility for recruitment, enrollment, scheduling, assessment, and project administration
DATA ENTRY	This role allows full permissions to assessment module for patients that the user has access to through another role. PHI access is not granted via this role.
DATA MANAGER	Data Managers: staff who need full access to data and functionality for the purposes of data entry, cleanup, and auditing
DEFAULT_PERMISSIONS	This role groups together default permissions that apply to all roles
GENETIC STAFF	Staff with access to genetic information
REFERRER	Allows read only access to patients and data without access to protected health information.
SYSTEM ADMIN	System Admin: staff who need full access to administrative functionality and read only access to data.
TESTER	A role to use for testing permissions

The DEFAULT_PERMISSIONS role is a special role that groups permissions that should be applied to all roles. These default permission may be configured to deny access to sets of actions. The default algorithm for determining if an action is permitted treats more specific "allow" permissions as overriding less specific "deny" permissions.

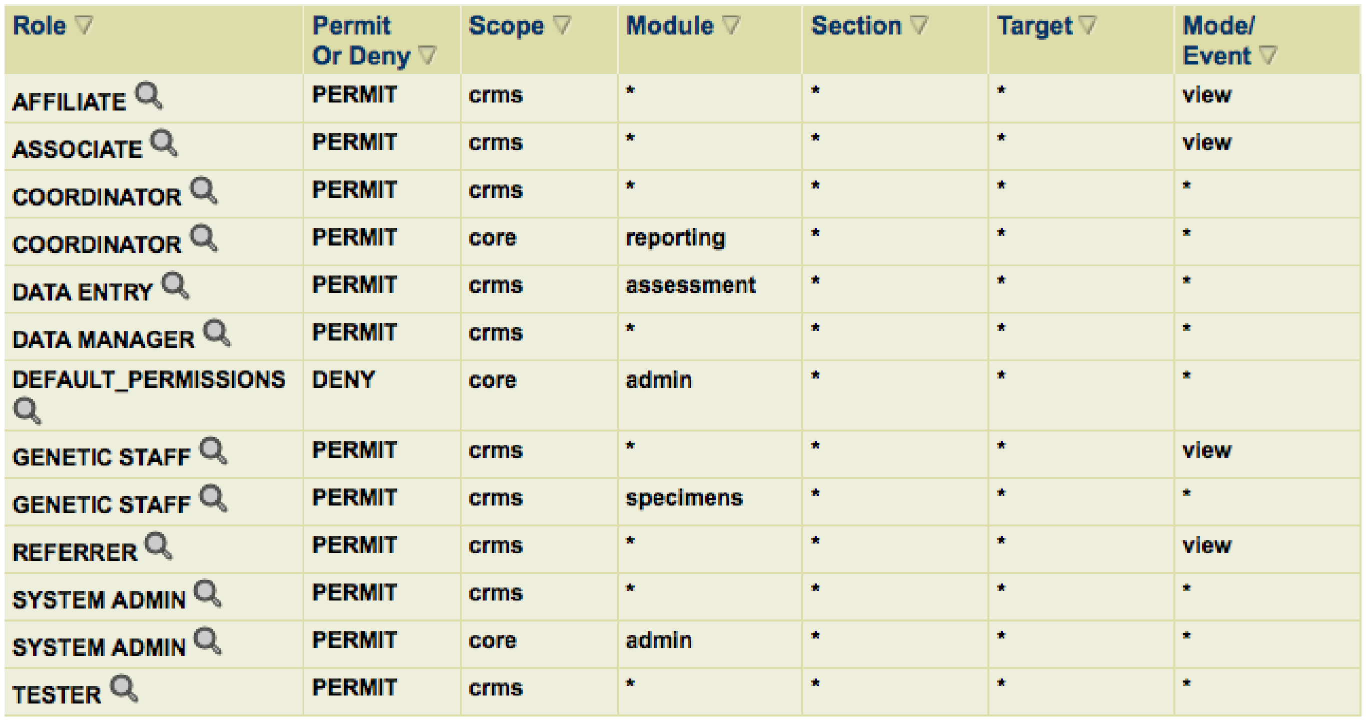
Roles can be configured through the LAVA user interface.

Permissions

Permissions in a LAVA application are associated only with roles and either allow or deny actions. Wildcards can be used to match multiple actions, and the algorithm used to determine whether a permission matches a particular action can be customized by LAVA application developers. Examples of permissions associated with the roles shown above are shown in the screen capture below.

In particular, notice the combination of the DEFAULT_PERMISSION's DENY for core.admin.*.*.* with the SYSTEM ADMIN role PERMIT permission for core.admin.*.*.*. The permit at the same or higher level of specificity allows the system administrators to access admin actions, while the default deny applies to all other roles.

Permissions can be configured through the LAVA user interface.



Role	Permit Or Deny	Scope	Module	Section	Target	Mode/ Event
AFFILIATE	PERMIT	crms	*	*	*	view
ASSOCIATE	PERMIT	crms	*	*	*	view
COORDINATOR	PERMIT	crms	*	*	*	*
COORDINATOR	PERMIT	core	reporting	*	*	*
DATA ENTRY	PERMIT	crms	assessment	*	*	*
DATA MANAGER	PERMIT	crms	*	*	*	*
DEFAULT_PERMISSIONS	DENY	core	admin	*	*	*
GENETIC STAFF	PERMIT	crms	*	*	*	view
GENETIC STAFF	PERMIT	crms	specimens	*	*	*
REFERRER	PERMIT	crms	*	*	*	view
SYSTEM ADMIN	PERMIT	crms	*	*	*	*
SYSTEM ADMIN	PERMIT	core	admin	*	*	*
TESTER	PERMIT	crms	*	*	*	*

Logging

Setup for logging is covered in the “LAVA Development Environment Installation” section. Assuming these instructions are followed, the log file is located in the Apache Tomcat logs directory and is called tomcat.log.

If you application will not load, or if you have encountered an error while running your application, the log file is one of the first things to check. In particular it can be useful to go to the very end of the log file and do a case sensitive backwards search for "ERROR" to find out where an error occurred.

However, depending upon where the error occurs in the application, there may not be an “ERROR” log entry, yet there may still be an indication of the error in the log file. This will require carefully scanning the log file entries at the time the error occurred. Experience will lead to having a better idea of where exactly to look when these types of obscured errors occur.

Java Debugging

As long as you start the Apache Tomcat server in Debug mode, the debugger will be enabled and you can set breakpoints in your LAVA source code and use the debugger.

However, if you want to debug into the libraries (jar files) used by the LAVA source code, you will need to take additional steps, to essentially tell the Java Virtual Machine (VM) that is running Apache Tomcat where the source code for these libraries resides on your computer.

Because all of the libraries used by LAVA are open source, the source code is freely available to download. Typically, the only open source libraries that you might need to debug are the Spring and Hibernate libraries, i.e. spring.jar, hibernate3.jar, maybe spring-webflow and spring-binding. Download and extract the source code for whichever library you need to debug.

After you get and unzip the source somewhere, you will need to edit the Launch Configuration for the Apache Tomcat server in Eclipse.

Go to the Servers tab, double-click the Tomcat server you set up, and then click on "Open Launch Configuration".

In the "Edit Configuration" window, "Source" tab:

Click Add../File System Directory and select the directory immediately above the package structure for the source code in hibernate, spring, spring-webflow,etc.. e.g.
../hibernate-3.2/src
../spring-framework-2.0.4./src
../spring-webflow-1.0.4/projects/spring-webflow/src/main/java

You may also be prompted for location of source during debugging and if you have the source, you can browse to the directory above the source tree at that time.

Production Release

This section covers the release of an application into production.

WAR file creation

The build.xml file has a "package-war" target which will be create a WAR file in accordance with the Java Servlet Specification. This single WAR file can be deployed in the Apache Tomcat "webapps" folder (in conjunction with a context file for your application in the Apache Tomcat "conf/Catalina/localhost" folder that will specify the database connection parameters for the application). This enables deployment of a production instance of your web application by transferring a single file to your production server rather than transferring the entire exploded directory structure of your web app.

The "package-war" target uses the packagedWar.file property in the build.properties file. This property should be set to the fully qualified war file which will be created when the build.xml package-war target is run. This target will create a WAR file in accordance with the Java Servlet Specification that can be copied to a production server for deployment. e.g.
packagedWar.file=/home/username/project/wars/demo.war

note: make sure that your development deploy directory contains the set of source that you want to run on your production server before creating the WAR file

Initial Release

For an initial production release, the database needs to be populated with the schema and initial data required by the application. This is typically done by generating a complete script from the development database and then running it on the production database. See the LAVA Relational Database section.

Database Schema Updates

Apply any database schema update scripts, if any, to your production database, to go along with the source code updates. Follow the instructions in any update documents. In particular, make a backup copy of the production database before updating it.

Metadata Updates

If there are metadata updates in any of the packages used by your application, run the latest metadata script from the LAVA CVS source repository on the production database. See Building, Deploying, Loading section for information on executing metadata scripts.

SSL

Generally, development is done using non-encrypted transmission of HTTP requests and responses. However, in a production environment, it is usually desireable to use encrypted transmission (i.e. https) via OpenSSL. Configuration of SSL is beyond the scope of this document, as it depends on the type of SSL certificate being used, e.g. temporary or permanent. However, we have used both at UCSF and can give you assistance if needed.

Creating a New App from the LAVA Demo App

For the purposes of this documentation, the application context path for the new web application will be "xyz", so the Eclipse project name will be "lava-app-xyz"

1) In Eclipse, in Package Explorer, right-click on the lava-app-demo project and choose Copy. Then right-click and choose Paste and the Copy Project dialog should appear allowing you to rename your project, e.g. lava-app-xyz

   This will change the project name in .project for you.

2) In Eclipse, in Package Explorer, right-click the new project (i.e. NOT lava-app-demo) project and choose Team/Disconnect... and in the confirmation dialog make sure you choose:

   "Also delete the CVS meta information from the file system."

3) Renaming Files and Folders to change "demo" to "xyz"

   Go thru all of the following folders and files and rename them, replacing the "demo" portion with your app, e.g. "xyz":

   database/local/demo

   images/local/demo/demo_logo.gif
   replace with your own logo .gif file, e.g. xyz_logo.gif

note: the images/local/demo/photoshop/DemoLogo.psd is a Photoshop file which can be used as a starting point in creating a logo for your app via Save As

   images/local/demo

   security/local/demo

   WEB-INF/context/local/demo/*
   rename the "demo" portion of every file in this directory, e.g. "xyz-context.xml"

   WEB-INF/context/local/demo

   WEB-INF/context/context-demo.xml

   WEB-INF/decorators/config/demo-decorators.xml

   WEB-INF/i18n/source/app-demo-* (e.g. "app-xyz-custom.properties")

   WEB-INF/jsp/local/demo

4) Editing File Contents

   a) Edit the security jsp files:
      security/local/xyz/login.jsp
      security/local/xyz/logout.jsp

security/local/xyz/retryLogin.jsp

In each of the above files:

- modify the HTML <title> text as desired
- replace demo in: images/local/demo/demo_logo.gif (e.g. images/local/xyz/xyz_logo.gif)
- modify the text beginning with "This demo application..." as desired

b) Edit the context files and replace "demo" with your context path (e.g. "xyz")

WEB-INF/context/context-xyz.xml
replace "demo" twice in "local/demo/demo-context.xml"
if your app does not use the lava-crms-nacc project then remove the import for “crms/nacc-context.xml”

WEB-INF/context/local/xyz/xyz-context.xml
replace "demo" in all the import resources

WEB-INF/context/local/xyz/xyz-home.xml
replace "demo" in <bean id="demo.crms.home.home.home"..

WEB-INF/context/local/xyz/xyz-scheduling.xml
If your app does not use lava-crms-nacc then delete configuration such that the “localVisitPrototypes” and “localInstrumentPrototypes” maps are empty, and completely remove the “initialUDS..”, “followUpUDS..”, “telephoneUDS..” and “ftldUDS..” beans

WEB-INF/context/local/xyz/xyz-security.xml
note: the "demo=demo,ROLE_USER" configuration is covered below under security and can be left alone for now

WEB-INF/context/local/xyz/xyz-reporting.xml
replace “demo” in:
demo.crms.reporting.reports.crmsReportLauncher
demoCrmsReportLauncherFormAction

note: WEB-INF/context/local/xyz/xyz-env-tomcat6.xml is discussed in the Database Connection section below

note: the following files can be ignored unless you are using the JBoss application server instead of Tomcat, or if you are going to use LDAP to authenticate users instead of configuring users and passwords in XML and the database:

/WEB-INF/context/local/xyz/xyz-env-jboss.xml would be used instead of xyz-env-tomcat6.xml

/WEB-INF/context/local/xyz/xyz-ldap-security.xml can be used in addition to xyz-security.xml

c) Edit the custom properties file.
WEB-INF/i18n/source/app-xyz-custom.properties replace all occurrences of "demo" (i.e. two on each line)

d) Edit jsp files
WEB-INF/jsp/local/xyz/crms/home/home/home.jsp
This is the home page.
- Modify the HTML title (including the "pageHeading" <meta> tag) as desired.
- Modify text starting with "This demo application.." as desired

e) Edit web.xml
WEB-INF/web.xml replace "demo" in 2 places:
/WEB-INF/context/context-demo.xml
/security/local/demo/redirect.jsp

5) Database Connection and Schema Creation

a) First of all, clean up the database directory to get rid of the scripts that are specific to demo data as follows:
- Remove scripts under database/local/xyz/development/complete
- Remove script under database/local/xyz/development/data, except for the 2 extract* scripts
- Rename extract-crms-app-demo-data.sql to extract-crms-app-xyz-data.sql, and edit the contents of both extract scripts, replacing all instances of “crms-app-demo” and “app-demo” with app-xyz, and replace demo with xyz in “call util_CreateMetadata…” (if you do not know your database name and database user yet, make a note to come back and edit extract.script when you do)
- Remove the folder and all of its contents: database/local/xyz/versions/older
- Remove the folders(s) and all of their contents: database/local/xyz/version/V3.* (so there is just a “versions” folder with no subfolders)

b) Create a database and user for your app in MySQL, e.g. database "lava_xyz", user "lava_xyz" and grant the user all privileges on the database.

```
mysql -u root -p
create database lava_xyz;
create user lava_xyz identified by '_PASSWORD_', lava_xyz@localhost
identified by '_PASSWORD_';
grant all on lava_xyz.* to lava_xyz, lava_xyz@localhost;
```

c) Create the schema in your database.

To start with an empty database, load the dump script in your database. The dump script is located in the lava-app-demo project. Use the dump script from the latest version of the database, under the "complete" folder. To start with an empty database, do not run the "with-ucsf-data" version.

Modify the location of your lava-app-demo project accordingly.

```
mysql -u lava_xyz -p lava_xyz < ~/workspace/lava-app-demo/database/local/demo/versions/V3.3/V3.3.1/complete/lava-demo-complete-uds-dump-3.1.0.sql
```

Also run any update scripts that have been created since the version of the dump script.

e.g. after dump script for V3.1.0 run the following scripts:
/lava-crms/database/crms/development/model/lava-crms-model-3.1.0-to-3.2.0.sql

And if creating a crms-nacc app, also run these scripts
/lava-crms-nacc/database/crms-nacc/development/model/lava-crms-nacc-model-3.1.0-to-3.2.0.sql
/lava-crms-nacc/database/crms-nacc/development/data/lava-crms-nacc-data-3.1.0-to-3.2.0.sql

d) Configure the Apache/Tomcat connection to your database.

Refer to the sample application context configuration file in lava-app-demo:
database/local/xyz/versions/V3.1/V3.1.0/config/demo.xml

In summary, you will do the following:

- make sure that your Apache Tomcat installation lib directory has a MySQL JDBC connector (see LAVA Development Environment Installation section)
- make sure that your Apache Tomcat installation lib directory has the jta-1.1.jar file (see LAVA Development Environment Installation section)
- create an Apache Tomcat context xml file for your web application, e.g. xyz.xml, with the proper configuration for your database connection, and put it in the Apache Tomcat deploy folder:
  conf/Catalina/localhost/xyz.xml
- in your app, lava-app-xyz, make sure the jndiName in WEB-INF/context/local/xyz/xyz-env-tomcat6.xml matches correctly with the Resource "name" in your Apache Tomcat context xyz.xml file, e.g.
  xyz-env-tomcat6.xml: java:comp/env/jdbc/xyz
  conf/Catalina/localhost/xyz.xml: jdbc/xyz

6) Security

The default configuration of the demo app, which has been cloned for your app, comes configured with two users, configured via XML. These are the "admin" and "demo" users configured as part of the Spring Acegi security module, in:

WEB-INF/context/local/xyz/xyz-security.xml e.g. demo=demo,ROLE_USER configures a username demo with password demo who can login into your app (there is also an admin user configured)

Each of these users has a record in the authuser database table. If you decide to modify the username or delete one of these users, you should make the corresponding changes in your application under the Admin module, Authorization section (if you just change the password you do not need to make any other changes).

NOTE: **** you will not be able to do this if you have deleted the pre-configured "admin" user, so if you want to delete "admin", you should create another user with Admin privileges in the LAVA Admin module/Authorization section first **** (the pre-configured “admin” user password is whatever is configured in the xyz-security.xml file)

In addition to authentication, this section is used to manage all authorization. See the “Authorization” section for more information.

Users that are configured in the above XML file have Auth Type "XML Config" in LAVA terminology.

In the Admin module/Authorization section, you can also create users with "LOCAL" Auth Type, where the user's password is stored in the database (encrypted). You will typically create this type of account for your users, as they can then change their own password in the Home module, User Info section of your app.

7) Modify build files

The build.xml (in the root directory of your app) file copied from lava-app-demo is setup to use the lava-core, lava-crms and lava-crms-nacc Eclipse projects. All applications will use lava-core, but if your application will not be using lava-crms or lava-crms-nacc, edit build.xml accordingly:

- if not using lava-crms-nacc, remove all lines referencing property "lava-crms-nacc.path"
- if not using lava-crms, remove all lines referencing property "lava-crms.path"

Edit build.properties and change the following:

a) deploy.path and warName are combined as follows to specify the location of your deploy directory, i.e. the directory under which the application source files from all Eclipse projects (e.g. lava-app-xyz, lava-core, lava-crms, lava-crms-nacc) are copied is a structure that is in accordance with the Java Servlet Specification and that will be deployed by Apache Tomact:

```
${deploy.path}/{warName}.war
e.g.
deploy.path=/home/username/project/deploy
warName=xyz
```

note: the requirement that the directory end in .war is a holdover from when JBoss was used

note: in the sample file:
database/local/xyz/versions/V3.1/V3.1.0/config/demo.xml
the docBase attribute that is mentioned in the comments would be set to this directory, e.g.
<Content docBase="/home/username/project/deploy/xyz.war/" ..>

b) instanceName should be set to your application context path, e.g.
instanceName=xyz

c) workspace.path should be set to your Eclipse workspace area, i.e. the folder above your Eclipse project folders, e.g.
workspace.path=/home/username/workspace

d) j2ee14lib.home should be set to the location of the Java Servlet Specification API library, i.e. servlet-api.jar e.g.
j2ee14lib.home=/usr/local/apache-tomcat-6.0.20/lib

e) packagedWar.file is the fully qualified war file which will be created when the build.xml package-war target is run. this target will create a WAR file in accordance with the Java Servlet Specification that can be deployed in the Apache Tomcat "webapps" folder (in conjunction with a context file for your application in the Apache Tomcat “conf/Catalina/localhost" folder that will specify the database connection parameters for the application)

e.g.
packagedWar.file=/home/username/project/wars/xyz.war

8) At this point you have completed the process of creating your initial application. You should now add it to a source control system. This will not be the same as the LAVA CVS repository, as that is limited to the lava-core, lava-crms, lava-crms-nacc and lava-app-demo (as well as other lava-app-* projects created by the UCSF development group). This means that your application will be using two source code repositories: the LAVA CVS repository to get updates for the LAVA projects that your app uses, and your own repository for your lava-app-* (and any other LAVA projects specific to your app).

This involves two things:

a) Setting up a repository in Eclipse. For CVS this can be done by switching to the "CVS Repository Exploring" perspective.

b) Adding your project to this repository. This can be done by choosing the Team menu for your project in Eclipse and choosing Share Project...

note: the build.properties file is development environment specific and should not be in your source control system unless there is only one developer or all developers have identical settings

You can now deploy and run your app. Consult the documents on installing Apache Tomcat and setting up your LAVA development environment, in order to deploy and run.

LAVA Framework

System Architecture and Requirements

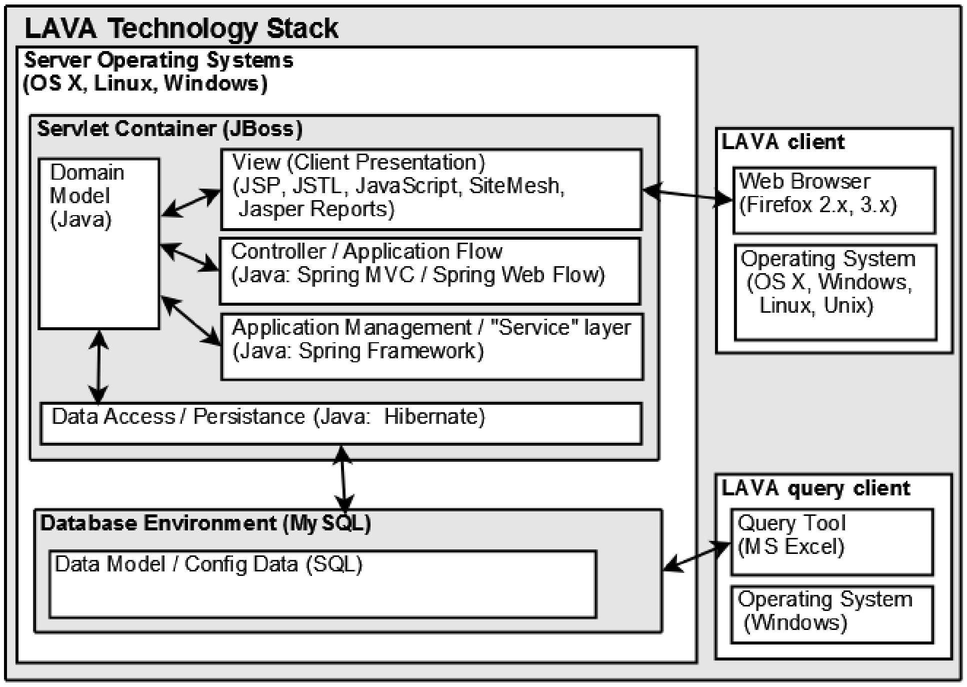
The LAVA framework is written primarily in Java with a mixture of languages and technologies used in the view layer components (JSP, CSS, JSTL, and Javascript). All current applications developed with LAVA run as servlets in the JBoss Application Server (JBoss AS) environment (although there is very little JBoss specific code in LAVA and using a simple Tomcat environment should be fairly straightforward to implement). Applications developed with LAVA have been run in production on OS X, Linux, and Windows server environments.

LAVA utilizes the Hibernate Persistence Library to de-couple framework and application code from any particular database environment. Existing LAVA applications have been developed using both MS SQL Server and MySQL, with the current open source version of LAVA using MySQL as the primary database environment.

LAVA was initially developed to support all major browsers. During development, official support for Internet Explorer was discontinued, with our primary focus on supporting all versions of Firefox (as a freely available cross-platform browser). The open source version of LAVA works seamlessly on Firefox and Safari (including iPhone), with only a few issues related to Internet Explorer that we anticipate could be easily resolved by anyone with motivation to do so. Our plans to integrate a standard javascript library like dojo/digit should resolve most of the Internet Explorer issues.

Architectural Highlights:

- LAVA application server is 100% Java and runs on any platform with Java support (Linux, OS X, Windows, Unix)
- Developed with robust, mature, open-source technologies, incl. JBoss, Spring Framework, Spring WebFlow, Hibernate, JSP, JSTL, SiteMesh, JavaScript, JasperReports
- Standard MVC (model-view-controller) design
- Tested with Firefox 2.x, 3.x web browsers on Windows, Linux, OS X.


LAVA Technology Stack
Server Operating Systems (OS X, Linux, Windows)
Servlet Container (JBoss)
Domain Model (Java)
View (Client Presentation) (JSP, JSTL, JavaScript, SiteMesh, Jasper Reports)
Controller / Application Flow (Java: Spring MVC / Spring Web Flow)
Application Management / "Service" layer (Java: Spring Framework)
Data Access / Persistance (Java: Hibernate)
Database Environment (MySQL)
Data Model / Config Data (SQL)
LAVA client
Web Browser (Firefox 2.x, 3.x)
Operating System (OS X, Windows, Linux, Unix)
LAVA query client
Query Tool (MS Excel)
Operating System (Windows)

Optimal / Minimum System Requirements

- LAVA Application Server
  - Servlet container (jboss, tomcat)
  - Server operating system (unix, os x, linux, windows)
  - Server class processor, 2 gb ram, 1 gb available storage
- Lava Database Server
  - MySQL 5.x
  - Any Operating System supported by MySQL 5.x
  - Sufficient processing, ram and storage capacity for serving the magnitude of application data and the number of concurrent users (highly application specific)
- Lava Client
  - FireFox 2.x, 3.x (other browsers work but are not “officially” supported)
  - PDF viewing software / plugins (for reports)
  - There are no specific operating system or hardware requirements.

Licensing

The specific license terms and conditions for LAVA are still under development. We are working on creating a free technology transfer agreement for non-commercial entities (licensees). LAVA is an open-source project, in that the source code is made available to LAVA licensees and that collaborative development and improvement of LAVA is a primary goal of sharing the framework.

Design Philosophy

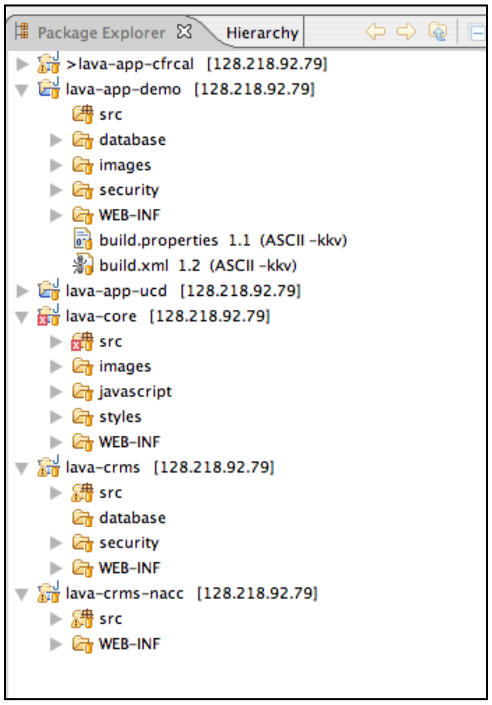
From the beginning, we understood that LAVA should support customization at almost every level of the architecture. The clinical research environment that LAVA was designed for is characterized by frequent change and innovation. When we were faced with tradeoffs between simplicity and extensibility in our design, we were generally biased toward complex solutions that enable simpler extensibility interfaces and methods.   We also tried whenever possible to anticipate how customizations might break when core functionality changed and took steps to avoid these side effects.

What this means in a practical sense is that if you need to customize LAVA in a way that we have anticipated then you will find it quite easy to do so; however, if you need to "look under the covers" to accomplish an unanticipated customization, it can get hairy quickly.

Logical Structure

There are a few core concepts to the LAVA logical structure that we have used to separate and organize functionality. The primary concepts are **project**, **scope**, **package** and **instance.**

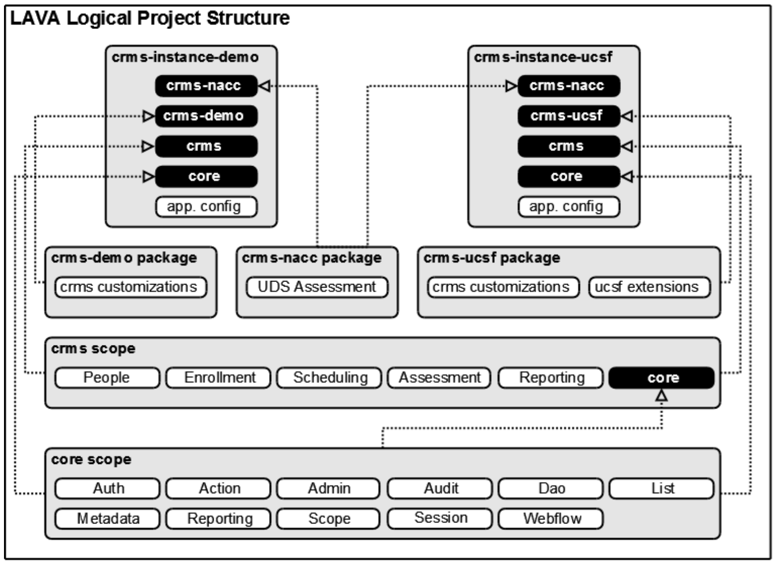
The primary development environment for LAVA has been the open-source Eclipse IDE. WE have organized the LAVA functionality into distinct Eclispe projects, and it is easiest to think of a LAVA project as roughly equivalent to an Eclipse project. Scopes, packages, and instances are defined in LAVA projects, and we have found that it is best to place each scope, package, or instance into a single project, and thus it makes sense to speak of a "scope project", a "package project" or an "instance project."   We use ant build tasks to combine the necessary code and resources from different projects into deployment directories (in war format).



```
▶ >lava-app-cfrcal  [128.218.92.79]
▼ lava-app-demo  [128.218.92.79]
      src
    ▶ database
    ▶ images
    ▶ security
    ▶ WEB-INF
      build.properties  1.1  (ASCII -kkv)
      build.xml  1.2  (ASCII -kkv)
▶ lava-app-ucd  [128.218.92.79]
▼ lava-core  [128.218.92.79]
    ▶ src
    ▶ images
    ▶ javascript
    ▶ styles
    ▶ WEB-INF
▼ lava-crms  [128.218.92.79]
    ▶ src
      database
    ▶ security
    ▶ WEB-INF
▼ lava-crms-nacc  [128.218.92.79]
    ▶ src
    ▶ WEB-INF
```

LAVA functionality (e.g. entities, actions, views) can be defined in scope, package, or instance projects. Low-level application services, and generic functionality that supports a specific class of applications are implemented within scope projects.   Package projects group functionality that may be used by more than one LAVA application. Functionality implemented in an instance project is only available to the specific application. Instance projects typically just implement customizations or extensions of functionality from other packages or scopes to meet the specific requirements of an application. Instance projects also contain configuration information about the application and also define (in the ant build.xml) what scopes and projects are combined in the instance.   The screen shot above shows the project explorer view of the Eclipse IDE. There are two scope projects shown (lava-core, lava-crms) one package project (lava-crms-nacc) and three instance projects (lava-app-demo, lava-app-ucd, and lava-app-cfrcal).

Every LAVA instance project must "include" at least one scope project. In practice we describe a LAVA application as "having" a particular scope--the highest level scope that it includes. As shown in the diagram to the right, scopes are organized hierarchically and "inherit" the functionality of their parent scope. At the base of the scope hierarchy is the **core** scope. The other LAVA scope that has been developed is the **crms** (clinical research management system) scope. As shown in the diagram below, the crms-instance-demo project includes functionality from the core scope, the crms scope, the crms-demo package and the crms-nacc package. The crms-instance-demo application is a crms scoped application. Note: The differences between the project names in the diagram below and the screen shot above reflect a change in naming convention that is still being implemented. The naming convention illustrated in the diagram below represents the current thinking about the best way to name LAVA projects



A number of extension mechanisms are introduced in the core scope that must have implementations in any scope that derives from core. These extensions use delegation and handler patterns to defer implementation details to derived scopes. In particular, each scope should define what entities define the current "context" of the application. For example, in crms scope applications, the "context" is defined by a current patient and/or project. For a payroll application, the context might be defined by current employee and/or department. A derived scope must also implement an authorization delegate that implements scope specific authorization checking logic (e.g. in crms, users are authorized to access patient records in the context of specific projects). Generally, a scope will also extend the core authorization entities to allow assignment of authorization roles to users for specific entities (e.g. 'this user is assigned the coordinator role for the ADRC research project' or 'this user is assigned the payroll manager role for a specific department'). Finally, scopes also help to define the default application flow. Each module and section within a LAVA application has a default action (the screen that is displayed when the user navigates to a new area of the application). Scopes may have different default actions depending on the current context of the application. For example, the default action in a crms application depends on whether there is a current patient selected.

An instance is best understood as a particular configuration of a web application running on the LAVA framework. This concept of instance is distinct from the more typical usage of "instance" to refer to any running process on a server or a specific server. When we use the term "instance" in this document, we are referring to our specialized meaning of instance. In other words, it would be reasonable to talk about any number of running "instances" (in the sense of processes) of a particular LAVA instance (in the sense of a specific customized configuration of LAVA functionality). A LAVA instance may best be thought of as collection of settings detailing what functionality is available to the users when the application is run, and what the applications looks like.

Once consideration is that functionality that is organized within a scope can be easily used and customized by particular LAVA instances, while incorporating or customizing functionality defined in one instance into another would be much more complex (and not recommended). This is because instances are based on a particular scope, and they "inherit" all the functionality of that scope (and higher level "parent" scopes), and can easily customize or hide scope level functionality.

Core Scope

The core frame scope provides functionality that is generic and applicable across different types of applications. This section provides a summary description of the functionality provided by each module in the core scope.

Action

Actions are the central organizing concept for application functionality in the LAVA framework. The action module defines base functionality for Action definitions and is closely coupled to the webflow and authorization modules.

Admin

The admin module primarily provides a user interface to system configuration/administration functionality including sessions and authorization.

Auditing

The auditing module provides a basic mechanism to track all changes to application data with a log of events, entities, and properties along with old and new values and the details of the user session responsible for the modifications (user, host, time).

Authorization

LAVA provides a standard role-based authorization module that enables all actions/events to be restricted based on the roles that are assigned to users and groups.

Dao

The Data Access Object (DAO) module layer in LAVA abstracts all aspects of data retrieval and persistence into a simplified object oriented interface. In addition, the DAO layer provides a standard mechanism for ad-hoc filtering of data by application users and for authorization based filtering that restricts records based on application specific data authorization rules.

List

The list module provides services for defining, customizing, and using lists of data values throughout LAVA applications. The module provides mechanisms for both static and dynamically populated lists.

Metadata
Metadata is used by the LAVA framework to define properties for controls and fields displayed in the user interface of LAVA applications.   Properties that are defined in metadata include the type of data, type of control to use (dropdown, textbox, text), the label for the control, required status, and lookup list.

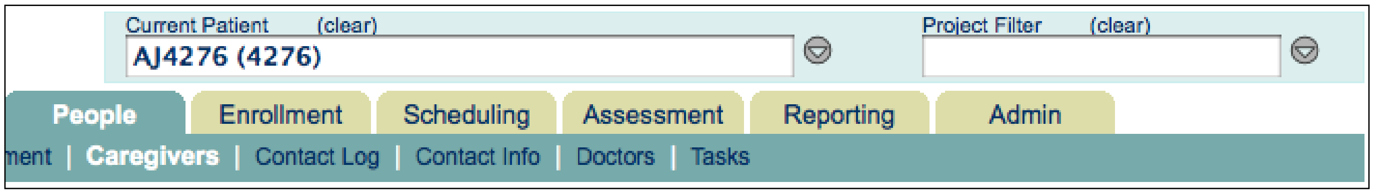
Reporting
The reporting module provides some basic reporting templates and a report launching screen with support for specifying basic report criteria (e.g. date ranges).

Session
The session module supports a logical LAVA session that is abstracted from the http session to track the user, host, time of login, latest activity, session expiration.

Webflow
The webflow module provides a programmatically-defined page flow management solution based on action definitions and their interrelationships. Flows with subflows ensure that context is retained in the parent flow when a subflow ends. Webflow has solutions for browser Back button issues and user double submits. Multiple flows can exist simultaneously and a user can resume any given flow from the state it was paused. Standard page flows for list display pages and entity CRUD (create, read, update, delete) pages are defined in the core scope. Custom page flows can be defined and applied to new action types.

CRMS Scope
The crms framework scope provides functionality and modules specific to clinical research projects such as patient management, enrollment management, visit scheduling, and assessment data collection. This section provides a summary description of the functionality provided by each module organized within the crms scope.

Patient / Project Context (not a module but important to explain here)
One of the primary features of a lava scope is the ability to provide a common set of organizing contexts for all functionality defined within the scope. For the crms scope, these organizing contexts are the Patient and Project entities.   What this means in practice is that all crms functionality is accessed in the context of a specific current patient and/or project. This enables data screens to be easily programmed to display data just for the current context and to limit data access based on the current user’s role assignments relative to the patients/project associations in the system.   The patient and project controls are shown in the screen shot below.



Assessment

The assessment module provides functionality for scheduling and entering assessment instruments (exam results, questionnaires, neuropsych, imaging, etc.) into LAVA applications. Standard instrument tracking fields and custom page flows for data entry and validation are also defined.

Enrollment

The enrollment module provides a standard mechanism for tracking patients/subjects relationship to projects or clinical programs over time. Consent form tracking is also provided.

People

The people module organizes functionality that relates directly to people, including Patient, Caregiver, Doctor, Contact Information, and Contact Log management.

Project

The project module provides functionality related to projects as a primary organizing structure within crms applications. Projects can be further segmented into units, each with distinct patient/subject populations and application users (this supports multi-site LAVA crms applications).

Scheduling

The scheduling module provides functionality for scheduling visits for patients/subjects in the context of specific projects.

Development of Applications with LAVA CRMS

This section presents an overview of LAVA CRMS application development in terms of the specific technical implementation layers.

Determining Where Application Functionality Fits in LAVA CRMS

When developing a new LAVA CRMS application, a typical early activity will be a fit/gap analysis of what functionality is required by the users of your application and the functionality already implemented in the CRMS scope. Identified gaps can be resolved through customizing existing functionality (e.g. adding new properties to the base Patient domain object), by implementing new functionality (e.g. adding a new domain object to manage waiting lists for appointments), or by implementing new instruments to capture the specific measures used to assess patients/subjects in a project.

While the specific implementation details for each of these approaches differ, the technical and programming constructs that you will work with are common to each approach and include domain objects, actions, flow types, handlers, jsp pages, metadata, etc.

Domain Model Objects

The LAVA framework has been implemented with a rich-domain model approach. By this we mean that LAVA model objects are not simply weak collections of properties, but rather feature-rich elements of the logical application architecture. The base LAVA model objects are coupled tightly to the DAO layer and mediate access to the DAO layer from other architectural layers through an embedded MANAGER object (specific and extensible for each model object) and via custom business rules defined as methods on the model objects.

In practice, this means that model objects know how to persist changes to themselves via a simple method signature `[modelObject.save()]` and can also retrieve instances of the model object through a simple static method signature `[ModelObject.MANAGER.getById(1234)]`. Business methods that apply to multiple instances of the model object are generally defined as methods on the static MANAGER object `[ModelObject.MANAGER.findOverdueObjects()]`, while methods that apply to a specific domain model instance are implemented as methods on the model class object itself `[modelobject.isOverdue()].`

All model objects are mapped to database tables via Hibernate Mapping (.hbm.xml) configuration files. Implementing a new model object is as simple as defining a new database table, implementing a java subclass of the base domain model class with appropriate properties for your database columns, and creating the Hibernate mapping file to link the two.

Action Definitions

Action definitions are the central organizing construct for functionality in a LAVA application. Each action has an action id that associates the action with an instance, a scope, a module, a section and a target (the action target). In addition, each action has a specific flow type that determines the kind of page flows built for the action. For example, the action used to modify the standard Caregiver model object is

```
      lava.crms.people.caregiver.caregiver
```

and is defined with the flow type of “entity.” The action above is translated as

```
      instance = lava  (lava is the standard instance identifier)
      scope = crms     (where the action is defined – crms scope)
      module = people  (found under the people tab in the UI)
      section = caregiver   (found under the caregiver section in the UI)
      target = caregiver    (target name for the action – must be unique)
```

To get slightly ahead of ourselves, the URL for this action would be:

http[s]://[server]:[port]/[instance name]/crms/people/caregiver/caregiver.lava

where [instance name] is the custom instance name for your application (e.g. 'demo' for the Demo application). Every web page rendered in a LAVA application is associated with a specific action definition, and the URL is generally the action id with '/'s substituted for the separators ('.') in the action id.

One mechanism for providing custom functionality in a LAVA application is defining an instance specific customization of existing functionality defined within a scope. This customization approach is initiated by defining an action that matches the action to be customized with the instance identifier changed from lava to your instance identified. For example to initiate a customization for the caregiver action described above for the 'demo' instance we would define the following action:

```
demo.crms.people.caregiver.caregiver
```

Once this action is defined, we can define custom code throughout the programming layers to implement the new functionality. We are jumping ahead slightly here, but this could involve

- Defining a new table in the database called `DemoCaregiver`
- Subclassing `edu.ucsf.lava.crms.people.model.Caregiver` to add properties for the new database columns in a new class `[org].[demo].lava.local.demo.crms.people.model.DemoCaregiver.`
- Implementing a new "subclass" hibernate definition file for `DemoCaregiver.`
- Defining a form action called `demoCaregiverFormAction,`
- Potentially defining new business functionality is a custom ComponentHandler subclassed from `edu.lava.crms.people.controller.CaregiverHandler`, and
- Making a modified copy of `jsp\lava\crms\people\caregiver\caregiver.jsp` in the `jsp\local\demo\people\caregiver` directory.

From this description of a customization process you can see that there are some standard customization conventions that we advocate based on prefixing the name of the code objects with the name of the instance. Following this convention makes it easy to know what is a customization and what is customized.

(Page) Flow Types

The page flow mechanism in LAVA is implemented using the Spring WebFlow library. What the Webflow module in LAVA provides is an implementation of generic flow type templates that can be programmatically generated for specific actions at runtime. The easiest way to explain this is with a detailed example, to stay consistent with the example above for action definitions, we will look at the flow type in the context of the primary caregiver action.

As described above, `lava.crms.people.caregiver.caregiver` action is assigned to the **entity** flow type. In practice we would simply say that it is an "entity action". When the LAVA application loads, is goes through a process of configuring actions and building flows. This is accomplished through the use of a `FlowTypeBuilder implementation ( see edu.lava.core.webflow.builder.FlowTypeBuilder ).`

The flow builder does a number of things, but primarily, it defines and registers flows (sets of pages, actions, transistions, etc) with a flow registry for the action. In our current example, the EntityFlowTypeBuilder would create the following flows for the Caregiver action:

- `lava.crms.people.caregiver.caregiver.add`
- `lava.crms.people.caregiver.caregiver.edit`
- `lava.crms.people.caregiver.caregiver.delete`
- `lava.crms.people.caregiver.caregiver.download`
- `lava.crms.people.caregiver.caregiver.view`

Let's look in more detail as the edit flow generated for the action (this is a simplified representation of the flow for the sake of clarity – there are many more things happening / possible in each generated flow):

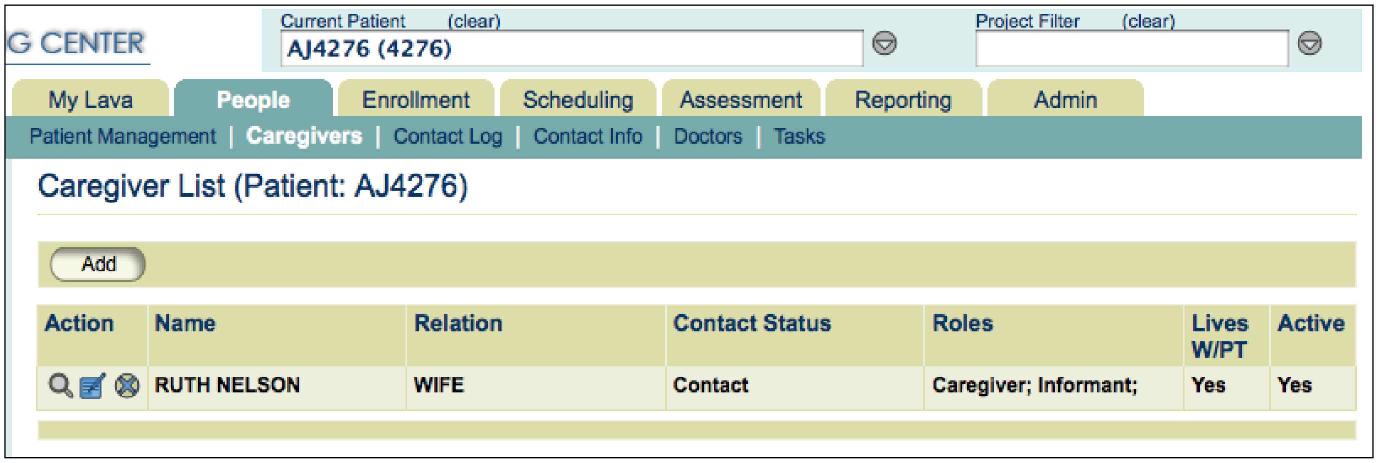
1. The flow begins with a FlowSetupState. This state does a lot of pre-rendering activity including retrieving the caregiver entity from the database and checking that the current user is authorized to edit the caregiver).
2. Then the flow transitions to the "edit" view state for the caregiver. This state presents a screen of editable properties for the specific caregiver record.
3. The edit view state supports a number of standard event transitions including "save" and "cancel".
4. The save event (triggered by clicking the save button in the user interface) binds the data from the form to the model object, validates the data, and if there are no problems saving the data, transitions to the finish state of the flow.
5. The cancel event (triggered by clicking the cancel button in the user interface) simply transitions to the finish state of the flow.
6. The finish state of the flow returns control back to the previous flow (for example the list of caregivers for the patient).

The connection points between flows (e.g. the transition from lists of data to editing data) is automatically configured by the FlowTypeBuilder based on the relationships defined between the action definitions.

Actions can be defined as parentFlows or subFlows of other actions. For example, the actual caregiver action definition looks like this:

```
<bean id="lava.crms.people.caregiver.caregiver"
parent="crmsEntityFlowAction">
      <property name="parentFlows">
           <list>
            <value>lava.crms.people.caregiver.patientCaregivers</value>
            <value>lava.crms.people.caregiver.projectCaregivers</value>
      </list></property>
</bean>
```

This definition configures the actions such that all the flows available for the caregiver action (add, edit, etc.) can be launched as subflows of the patientCaregivers or projectCaregivers action flows.   What this translates into in the user interface is that you can view, edit, or delete the caregiver records displayed in lists of caregivers as shown in the screen capture below



Caregiver List (Patient: AJ4276)

Action	Name	Relation	Contact Status	Roles	Lives W/PT	Active
	RUTH NELSON	WIFE	Contact	Caregiver; Informant;	Yes	Yes

There are no predefined limitations to the complexity of page flows or relationships between actions or flow type in LAVA. As of this writing, the primary Flow Types available to construct functionality are entity, list, instrument, instrument list, and report.   Most standard application functionality can be expressed in terms of entities and lists of entities.

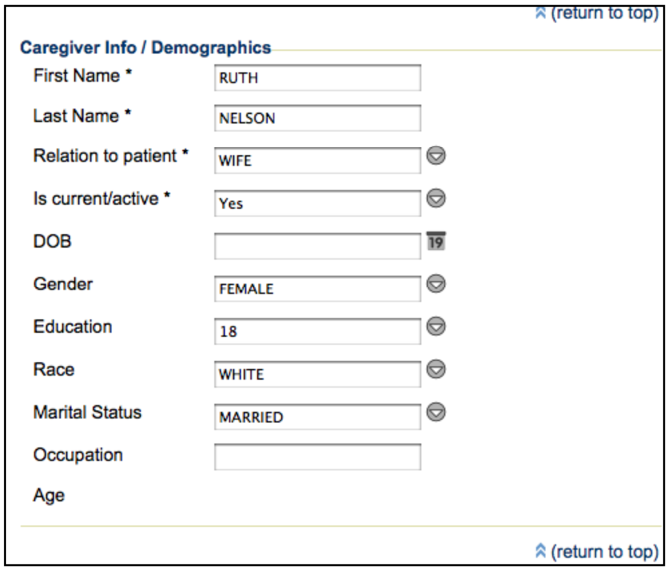
Form Actions and Component Handlers

Most action definitions in a LAVA application have a corresponding Form Action definition and at least one ComponentHandler. A ComponentHandler is a java class where code is written to handle the events associated with the action's flow definition. A simple way to think about this is that every button clicked on the user interface to submit a page is eventually handled in an event handling method defined on a ComponentHandler class. The standard behavior for most functionality (saving, deleting, etc) is already coded into standard ComponentHandlers that can be easily subclassed for new entities.

The LAVA architecture supports rendering multiple entities or multiple lists of entities or combinations of the two on single web pages. This is handled by configuring multiple ComponentHandlers in the FormAction for the action. In this context, each entity or list is thought of as a "component" on the page, thus the naming convention of calling the handler class a "ComponentHandler." The form action for the caregiver action we have been using in our examples is:

```
<bean id="caregiverFormAction" parent="crmsEntityComponentFormAction">
        <constructor-arg><list>
                <bean class="edu.ucsf.lava.crms.people.controller.CaregiverHandler" parent="crmsHandler"/>
        </list></constructor-arg>
</bean>
```

From this example, you can see that form action bean names are constructed by taking the target of the action id and adding a suffix of "FormAction," in this case "caregiverFormAction". Form actions take a list of Component Handlers as a constructor argument. In this example, there is only one handler that we need to support the caregiver action. It is appropriately named the "CaregiverHandler" and is located is in the edu.ucsf.lava.crms.people.controller java package. The convention is to group all handlers for a module into a single controller package for that modules.


(return to top)

Caregiver Info / Demographics

First Name *	RUTH
Last Name *	NELSON
Relation to patient *	WIFE
Is current/active *	Yes
DOB	
Gender	FEMALE
Education	18
Race	WHITE
Marital Status	MARRIED
Occupation	
Age	

(return to top)

The base classes for Form Actions and Component Handlers that are used to write application specific functionality contain standard implementation methods for most of what is required to support the rendering of the pages and the handling of events.

View Layer

The view layer (web pages) in a LAVA application are currently written using a combination of jsp and custom JSTL tags. Our philosophy has been to abstract out the actual rendering of html from the writing of LAVA view pages. We accomplish this using the SiteMesh library to handle rendering the page layouts for different types of pages (entity, list) and for different components on the page. We have also written a large number of JSTL tags (or macros) that are used to render data fields, lists, listrows, buttons, etc. Finally there is a custom set of tags written to support complex on screen skip logic (e.g. when the value of one control is equal to a certain value then disable another set of controls).

The most important JSTL tag in LAVA applications is <tags:createField /> . The createField tag looks up information configured in the system metadata to render the appropriate control for the specified property and to bind the data from domain objects backing the forms. A simple example of this tag is visible in the code snippet below from the caregiverContent.jsp that renders the page section shown in the screen capture to the right.

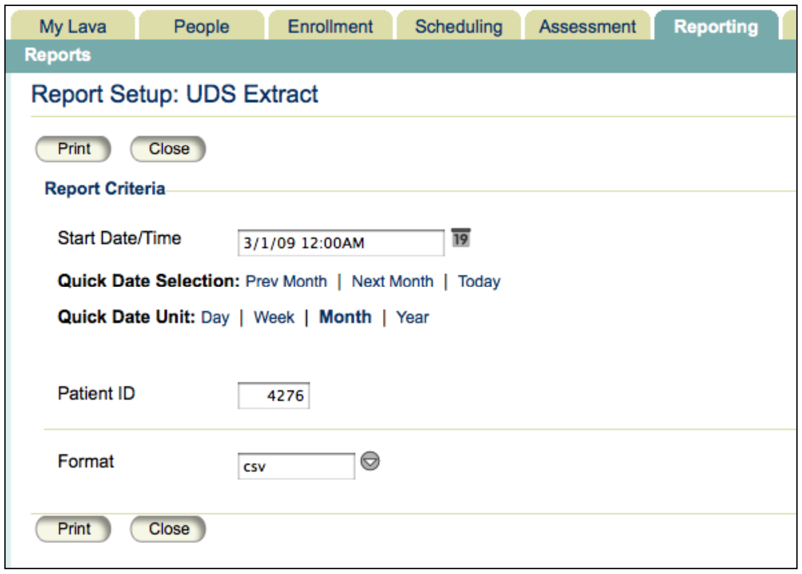
```
<page:applyDecorator name="component.entity.section">
	<page:param name="sectionId">caregiver</page:param>
	<page:param name="sectionNameKey">caregiver.caregiver.section</page:param>
	<page:param name="quicklinkPosition">top</page:param>

	<tags:createField property="firstName" component="${component}"/>
	<tags:createField property="lastName" component="${component}"/>
	<tags:createField property="relation" component="${component}"/>
	<tags:createField property="active" component="${component}"/>

	<tags:createField property="birthDate" component="${component}"/>
	<tags:createField property="gender" component="${component}"/>
	<tags:createField property="education" component="${component}"/>
	<tags:createField property="race" component="${component}"/>
	<tags:createField property="maritalStatus" component="${component}"/>
	<tags:createField property="occupation" component="${component}"/>
	<tags:createField property="age" component="${component}"/>
</page:applyDecorator>
```

This snippet first applies a SiteMesh decorator to generate the HTML for a section of properties on the resulting web page. This decorator generates an appropriate section heading and adds a link back to the top of the page. Within this section, calls to the createField tag render data entry controls for the caregiver properties as shown in the screen capture above.

There are no predefined limitations to the view layer technologies you can use with LAVA. We encourage implementation of simplified macros or tags to abstract the details of rendering complex view layer markup.

Reports

Reporting functionality is the LAVA framework is implemented using the open source JasperReports libraries. There is a standard report launching action and flow that allows developers to create links to launch reports from anywhere in the interface and redirect the user to standard screens for capturing report filter criteria (such as date ranges) before generating the report as shown in the screen capture below.



Report Setup: UDS Extract

Print Close

Report Criteria

Start Date/Time: 3/1/09 12:00AM

Quick Date Selection: Prev Month | Next Month | Today

Quick Date Unit: Day | Week | Month | Year

Patient ID: 4276

Format: csv

Print Close

Any functionality that is supported by JasperReports can be used with LAVA applications, however, we encourage developers to supply data to reports via the lava domain model rather than through direct jdbc connections to the database (this ensures that the built in data authorization filtering is applied to the reports and that users will not be presented data that they have not been authorized to view).

Some LAVA Terminology Defined

In this section, terminology that developers should be familiar with or that have LAVA specific meanings are defined.

Project is an organizing concept for grouping functionality. At this time, a "logical" projects pretty much correspond to a "physical" eclipse project. There are three kinds of projects, scope projects, package projects, and instance projects.

LAVA functionality (e.g. entities, actions, views) can be defined in scope, package, or instance projects. Low-level application services, and generic functionality that supports a specific class of applications are implemented within scope projects.   Package projects group functionality that may be used by more than one LAVA application. Functionality implemented in an instance project is only available to the specific application.

Scope is a hierarchical organization of functionality. Scopes define low-level application services and generic functionality that supports a specific class of application.   Every LAVA application has a scope, and can include or extend functionality from that scope and any of the functionality implemented in the higher level "parent" scopes.

Package is a "horizontal" organization of functionality. Packages allow functionality to be grouped based on arbitrary criteria (e.g. these assessment measures are all defined by a particular organization).   Generally packages group functionality from just a single scope and can be included (used) by multiple application instances.

Instance is an application specific organization of functionality. Instance projects typically just implement customizations or extensions of functionality from other packages or scopes to meet the specific requirements of an application. Instance projects also contain configuration information about the application and also define (in the ant build.xml) what scopes and projects are combined in the instance.

Module refers to a grouping of related functionality. These modules can either group functionally related actions (e.g. auditing, sessions) or can group process related functionality (e.g. enrollment, assessment).   Generally, if a module's functionality is represented within the user interface of a LAVA application, it will be assigned a tab in the navigation bar.

Section refers to a grouping of functionality more specific that the level of module. Generally, if a section's functionality is represented in within the user interface of a LAVA application, it is listed a one of the sub-navigation links on the navigation bar.

Action represents a specific unit of functionality within a LAVA application. The components of an action are structured as [instance].[scope].[module].[section].[target].

Mode represents an entry point into functionality provided by an action. For example, an entity action supports the add, view, edit, and delete modes. These modes are appended to the action to form a flow id.

Flow is a Spring WebFlow construct and there is one flow for each combination of Action and Mode, e.g. lava.crms.scheduling.visit.visit.edit

Extending and Customizing

Once a LAVA application has been created from the LAVA Demo app, another app or from scratch, the application will more than likely need to extend and customize LAVA. This section covers many of the most common ways to modify a LAVA application.

The examples in this section are based on a fictitious application with a Java Servlet Specificaiton application context path of “xyz”. This context path string is referred to as the application instance in this guide.

Instance and Scope

The notion of instance and scope are vital to the customization of LAVA. Both of them are represented as database column values, parts of Java class package names, and parts of web application directories. It can be difficult to completely understand the usage of one vs. the other, but the examples in this section will clarify that.

Instance represents things that are specific to the application. There is a notion of a default instance value of “lava” which represents the default, whether it be metadata or an action, regardless of scope. For example, all of the metadata and actions in the inner scopes “core” and “crms” have default definitions where instance is defined to be “lava”, and an application can then extend or override those definitions using its own instance, e.g. “xyz”.

Scope distinguishes classes and metadata as belonging to a particular scope, and thus belonging to a particular LAVA project, e.g.
lava-core project: ‘lava’ instance, ‘core’ scope
lava-crms project: ‘lava’ instance ‘crms’ scope
lava-crms-nacc project metadata has ‘lava’ instance ‘crms-nacc’ scope
lava-app-xyx metadata has ‘xyz’ instance and ‘app-xyz’ scope

This distinction is vital to ensure separate namespaces among scopes, and to manage metadata in terms of the source code repository, and doing releases to a production server.

The best way to understand the usage of instance and scope is in the practical application of extending and customizing LAVA , covered in the remainder of this section.

Customizing the Interface

Color Scheme

The color scheme used by LAVA can be changed by modifying all of the color codes used in the lava-core/styles/styles.css CSS stylesheet. Comments have been added to the stylesheet to help make this an easier process involving search and replace. Follow the instructions at the top of styles.css.

Note that there are several sample color schema stylesheets in lava-core/styles that are for reference purposes only. These stylesheets have not been kept up to date with respect to the changes made to styles.css so they can not simply be plugged in.

Logo

The logo is placed at the top left of every page. The logo file is a .gif file and LAVA has a built-in naming convention where the logo file for the “xyz” app must be:
lava-app-xyz/images/local/xyz/xyz_logo.gif

The LAVA demo app provides a Photoshop template file for creating a logo image with the property size and properties and this can be found in the “photoshop” subdirectory of the “images/local/demo” directory.

Login Pages

To modify the content of the login and logout pages, modify the files in this directory:
lava-app-xyz/security/local/xyz

Welcome Page

To modify the content of the Welcome page, modify the following file:
lava-app-xyz/WEB-INF/jsp/local/xyz/crms/home/home/home.jsp

Footer

To modify the footer at the bottom of each page (note that the default footer is currently empty):
Modify lava-app-xyz/WEB-INF/i18n/source/app-xyz-custom.properties

Add the following lines:
xyz.footerURL=local/xyz/navigation/footer/footer.jsp
xyz.modalFooterURL=local/xyz/navigation/footer/modalFooter.jsp
where this location is appended to /WEB-INF/jsp/ to generate the location of the jsp.

Then create the jsp files with the desired content. Note this overrides the use of the defaults:
lava-core/WEB-INF/jsp/navigation/footer/footer.jsp
lava-core/WEB-INF/jsp/navigation/footer/modalFooter.jsp

Metadata

LAVA maintains metadata for each viewable property in the database which contains display characteristics for the property. This includes:

- the label for the property
- how the property is displayed in readonly mode, e.g. when just viewing the data
- what form control is used for the property in data entry mode
- the size of the control used for data entry
- the maximum length of text allowed for string properties
- if a property should be visually marked as required

- for properties that have lists, e.g. autocomplete dropdown boxes, the metadata designates the list that populates the list
- HTML attributes for the property form control, including Javascript event handlers

This metadata can be modified to customize these display characteristics of a given property. General information about metadata can be found in the Metadata section.

The viewproperty table has “instance” and “scope” columns. The usage of the “instance” and “scope” columns will be covered in the customization examples that follow.

The ViewProperty table columns which control the display related characteristics of a property are as follows:

- “context”
- “style”
- “required”
- “label”
- “label2”
- “maxLength”
- “size”
- “indentLevel”
- “attributes”
- “list”
- “listAttributes”
- “propOrder”
- “quickHelp”

Descriptions follow:

- the "style" metadata value for the property style of the property for which a field is being created, where the style represents the nature of the values that the property can assume, i.e. style represents several structures which characterize the values which a property can assume, as follows:
    - "scale" = list of less than N possible values, where N = ?
      technically split between continuous (e.g. 1..10) and non-continuous (e.g. 0,0.5,1,2,3,4), but for our purposes, based on the maximum number of possiblevalues N
    - "range" = list of greater than N possible values, where N = ?
    - "suggest" = a list of suggested values, a user supplied value is acceptable.
    - "string" = value less than 50 chars
    - "numeric" = numeric value, treated same as "string" except right aligned
    - "text" = value greater than 50 characters
    - "date" = date value
    - "time" = time value
    - "toggle" = boolean value
    - "multiple" = list from which multiple values can be selected (the data is bound as a comma-separated string)

A given "style" can result in different presentation based on the current mode.

- “context" metadata value describes the context of the property
    - "c" = contextual data, i.e. readonly display data, e.g. the current patient name
    - "i" = informational data, e.g. instrument data collection quality issues
    - "r" = result data, i.e. instrument data
    - "h" = hidden field, e.g. a hidden input.

    Notes:
    - “r": for some styles, result context is used to determine which input control to use, e.g. for style=scale, autoComplete is used, but in collect mode ('dc'), a comboRadioSelect is used.
    - "r" fields generally correspond to the requiredResultFields list of instrument entity classes. however, this list is created dynamically in realtime so the actual requiredResultFields may be a subset of fields with context "r"
    - “r" vs. "i" for instrument double entry: subject to the previous note, all "r" fields are part of the double entry comparison, whereas "i" fields are not however, because of uitags skip logic which may involve "i" and "r" fields, "i" fields must still be present on the "doubleEnter" page even though they are not compared, and on the "compare" page they must have both the first and second entries present (again, for uitags purposes) but since they are not compared, the second entry is hidden, to make things clear to the user

        *** therefore, an "i" field should not be part of an instrument's requiredResultField list because then would have situation where the field is compared but the field's second entry on the "compare" page is hidden additionally, all result context fields have two input fields on the instrument "enter" flow, compare page (whereas context "i" fields only have one, i.e. for fields that should never be involved in the compare), with the exception of those with the "disabled" within the value of the “attributes” metadata, whch is typically a non-editable total and should only appear once on double enter compare

- "section" generally not used in the display of properties on a form, but it can be used when using code generation utilities to generate the Jasper Report to print an entity record, to divide the properties into sections
- "required" whether the property is a required field, i.e. purely whether the property label should visually indicate that the field is a required field ("Yes" or "No"). the enforcement of required fields is done in the controllers/handlers and is completely independent this metadata field.
  note: this is ignored for content “r” fields, i.e. those used by instruments, because instrument fields are generally always required, even if the value input is a skip code or standard error code
- “label" is a String of the label to use for the field
- "label2" is a String of a second label that can be used for the field, e.g. the data dictionary variable name used for the field
- "maxlength" used to set the max length of text in fields where user can type text, i.e. style=string, suggest, text. one exception is select controls, where maxlength is the number of options to display.
- "size" set the size of fields that use an HTML text box, i.e. style=string, numeric, date, datetime, scale, range, suggest, select, multiple

note: the size of style=”text” fields can be set using rows and cols in the attribues column metadata, or alternatively, by creating a style with sizing and passing it into the dataStyle attribute of this tag, e.g. the "instrNote" style
- "indentLevel" an integer (as String) the indent level for the field, where "0" means do not indent, "1" means indent one level, etc.
  e.g. if the field is question 3a. it should be indented from question
  3 and its indentLevel=1
- “attributes" for fields represented by an input control, the attributes as a String that are used as HTML attributes for the input HTML tag, e.g. the HTML attributes for a textarea: rows="14" cols="40"
- “list" the name of the list for fields represented by an input control that use a list. the name is used as the key into the "staticLists" or dynamicLists" Maps in the model to obtain a list which is a Map<String,String> structure where the Map entry keys are the list item values and the Map entry values are the list item labels
  note: lists are defined as beans in XML and all have the “list.” prefix. This prefix is omitted when specifying the list to be used in metadata
- "listAttributes" additional attributes that modify the creation of the list specified in "list"
- "propOrder" not used in the display of properties on the form (they are displayed in the order they appear) but it can be used when using code generation utilities to generate the Jasper Report to print and entity record
- “quickHelp" intended for tool tips to give a description of the property beyond the label

The JSP tag createField (lava-core/WEB-INF/tags/createField.tag) uses these metadata values in conjunction with the display “mode” to ultimately determine who a property is displayed on a web form.
The possible display “modes” are:

“dc” data collection mode
The UI is geared toward a direct collect mouse or touch driven mode, e.g. radio buttons are used instead of a dropdown so all options are visible

“de” data entry mode
The UI is geared toward rapid keyboard entry of paper forms, especially with numeric code values, where a user quickly keys in a number, tabs to the next property, keys in the next number, etc.

“vw” view mode
This is the readonly mode used when viewing an entity record. Generally no controls are used, the value of each field is displayed along with the field label.

“le” list edit mode
This is for an editable list, which is not implemented yet in LAVA

“lv” list view mode
This is the mode for regular (readonly) display of a list of entities.

Note that a view can pass in a “mode” as an attribute to the createField tag to override the currently set “mode”.

In addition to the display “mode” used to display fields, there is also a display “view” value that views can utilize to include logic as to how to display a given web form. The possible values for display “view” are the standard CRUD operations:

- “view”
- “edit”
- “add”
- “delete”

For a list, the display “view” would be set to “view” (when editable lists are implemented in LAVA then “edit” will also be possible for lists.

Application Specific Properties

Whether an application extends existing LAVA entities by adding additional properties, or adds new entities altogether, each application property needs a metadata record in the viewproperty table to designate how each property should be displayed. The viewproperty scope column for these custom properties will be the scope used for the application, e.g. ‘app-xyz’

This will ensure that your application specific properties are kept separate from properties in other scopes. For example, if you used scope ‘crms’ for your application properties because they are Clinical Research properties, and this was committed to the source code repository for the lava-crms project, the metadata for your properties would end up being part of every application that uses ‘crms’ scope, which is clearly not what was intended.

Additionally, the scope is used when creating the metadata script for your app, using this script:
database/local/xyz/development/data/extract-app-xyz-data.sql
This generates a script with all of the property metadata and list data that is custom to your application. It is useful both for multi-programmer development as a way to sync metadata changes between developers, and for application releases to transfer this data from the development database to the production database. See the Metadata section of this document for more information.

For example the following metadata record could be created:
messageCode=*.patient.mrn
locale=en
instance=lava
scope=app-xyz
entity=patient
property=mrn
context=i

style=string
required=No
label=MRN
maxLength=7
quickHelp=Medical Record Number

NOTE: for the addPatient action, because the command class is a DTO, there is a level of indirection between the command object and the Patient object, so the property needs to be prepended by 'patient_' and the above would be modified as follows:
messageCode=*.addPatient.patient_mrn
entity=addPatient
property=patient_mrn

Property Metadata Override

To override the metadata for an existing property, set the instance column to the application instance, e.g. "xyz" and use the same values for messageCode, locale, scope, prefix, entity and property. Then modify the remaining values as desired.

For example, to override the "subjectStudyId" property of the "enrollmentStatus" entity in the 'crms' scope to give it a new label of "Enrollment ID" instead of "Subject Study ID", you would insert a row into viewproperty that is identical to the existing row in viewproperty for entity=enrollmentStatus, property=subjectStudyId with the following changes:

instance=xyz
label=Enrollment ID

Customizing Lists Based on Project

There are a number of lists used by lava-crms that have the ability to be customized based on the Project context. This list customization is for forms which have a Project property in conjunction with another property on the form which is populated with a list that is dependent upon Project.

This section does not cover the details of defining lists. List definition is discussed in more detail in the "List Definition" section.

Values can be added to the list as "GENERAL" in which case they are always part of the list, regardless of the current Project. Values can also be added to the list in association with a specific Project, so that they are only part of the list when that Project is the current Project.

The following lists can be customized in this way:

- VisitType for the Visit entity, visitType property

- VisitLocations for the Visit entity, visitLocation property
- StaffList for the Patient entity, createdBy property, the Visit entity, visitWith property and others
- ConsentType for the Consent entity, consentType property
- ReferrralSources for the EnrollmentStatus entity, referralSource property
- ProjectStatus for the EnrollmentStatus entity, status properties

StaffList
StaffList differs from the others in that the defined lists are created by combining the users in the system with the users in the StaffList list. In other words, the StaffList list is for adding staff members to the list who are not users of the LAVA application. Admin users are excluded from these lists.
Note that in addition to "list.all.staffList" which is the union of all (non-System Admin) users in the authUser and the users in StaffList (Project dependent), there are also "list.patient.patientStaffList" and "list.project.projectStaffList" which consult the user authorization permissions in constructing the staff lists:

"list.patient.patientStaffList" includes all users in authUser that have permissions in projects in which the current patient has an enrollment combined with all users defined in StaffList where the project is one in which the patient is enrolled or the project is "GENERAL".
i.e. this is used in conjunction with the current patient context and would not make sense when there is not a current patient, e.g. in Add Patient

"list.project.projectStaffList" includes all users in authUser that have permissions in the current project context combined with all users defined in StaffList where the project matches the current project context or the project is "GENERAL".

ProjectStatus also differs from the others and is covered in Customizing Enrollment Statuses below.

As an example, let's customize the VisitType list for a project called "ABC Study", giving it the following VisitTypes:

- "Baseline"
- "1 Year"
- "2 Year"

Along with the "Specimen" Visit Type which is available for all projects.

The following records would be inserted into the listvalues table:

```
INSERT INTO `listvalues`
('ListID', 'instance', 'scope', 'ValueKey', 'ValueDesc', 'OrderID')
SELECT 'ListID', 'lava', 'app-xyz', 'ABC Study', 'Baseline', 1)
FROM 'list' where 'ListName' = 'VisitType';
```

```
INSERT INTO `listvalues`
('ListID', 'instance', 'scope', 'ValueKey', 'ValueDesc', 'OrderID')
SELECT 'ListID', 'lava', 'app-xyz', 'ABC Study', '1 Year', 2)
FROM 'list' where 'ListName' = 'VisitType';

INSERT INTO `listvalues`
('ListID', 'instance', 'scope', 'ValueKey', 'ValueDesc', 'OrderID')
SELECT 'ListID', 'lava', 'app-xyz', 'ABC Study', '2 Year', 3)
FROM 'list' where 'ListName' = 'VisitType';

INSERT INTO `listvalues`
('ListID', 'instance', 'scope', 'ValueKey', 'ValueDesc', 'OrderID')
SELECT 'ListID', 'lava', 'app-xyz', 'GENERAL', 'Specimen', 100)
FROM 'list' where 'ListName' = 'VisitType';
```

Note: set OrderID to 100 so this will be the last item in the list for all VisitType lists

By the way, the visit.visitWith property is an example where the style column in viewproperty is 'suggest'. What this means is that the user is not restricted to picking items from the list. The list is just a list of suggested values, but the user can type their own value into the list. This does not add the value to the list itself, but the value is stored in the database so it is added to the list whenever that particular entity is retrieved.

Customizing Enrollment Statuses

Customizing Enrollment Status follows the same guidelines as customizing other entities in the system in terms of extending the base class entity and handler and defining an instance-based action. However, Enrollment Status has an additional element of customization having to do with the statuses that are used.

If you need to customize the Enrollment Status handling and/or views in addition to project statuses, you should reference the “Visit and EnrollmentStatus Entity Instance Customization” section or “Visit and EnrollmentStatus Entity Dynamic Customization” section in conjunction with this section.

Enrollment Status contains a series of status values representing all possible statuses that a patient could have for a given project. Because these values can be customized, they are persisted with each Enrollment Status record, so that they can then be used as Status values in the Enrollment Status view. The enrollmentstatus table has a series of three columns for each status, e.g. ReferredDesc, ReferredNote and ReferredDate. It is the *Desc column that stores the status value, known as ProjectStatus for each status. Note that the columns are named after the default ProjectStatus values, and so when a ProjectStatus value is customized with a different status, there is a mismatch, e.g. the EligibleDesc column could store the custom status “PRE-APPOINTMENT” instead of the default “ELIGIBLE”.

The default EnrollmentStatus ProjectStatus values are hard-coded in the edu.ucsf.lava.crms.enrollment.model.EnrollmentStatus class

If a project needs to use custom ProjectStatus values, the following modifications need to be made.

1) The ProjectStatus values must be configured in the localEnrollmentStatusPrototypes bean, which is in one of the files included in the application WEB-INF/context/context-xyz.xml file, e.g. WEB-INF/context/local/xyz/xyz-local.xml

   For example, using the project "AUTOPSY":

   ```
   <bean id="localEnrollmentStatusPrototypes" class="java.util.LinkedHashMap">
   <constructor-arg><map>
   <entry key="Autopsy">
           <bean class="edu.ucsf.lava.crms.enrollment.model.EnrollmentStatus">
                   <property name="referredDesc" value="REFERRED"/>
                   <property name="deferredDesc" value="DEFERRED"/>
                   <property name="eligibleDesc" value="UNDER_CONSIDERATION"/>
                   <property name="ineligibleDesc" value="INELIGIBLE"/>
                   <property name="declinedDesc" value="DECLINED"/>
                   <property name="enrolledDesc" value="ENROLLED"/>
                   <property name="excludedDesc" value="EXCLUDED"/>
                   <property name="withdrewDesc" value="WITHDREW"/>
                   <property name="inactiveDesc" value="ENROLLMENT_PENDING"/>
                   <property name="deceasedDesc" value="DECEASED"/>
                   <property name="autopsyDesc" value="AUTOPSY_PERFORMED"/>
                   <property name="closedDesc" value="NOT_PERFORMED"/>
           </bean>
   </entry>
   </map></constructor-arg>
   </bean>
   ```

   Note: if EnrollmentStatus is extended, e.g. XyzEnrollmentStatus, then that fully qualified class name should be substituted for edu.ucsf.lava.crms.enrollment.model.EnrollmentStatus above.

   Note: not all of the ProjectStatus columns provided by the enrollmentstatus table need be used, e.g. to exclude a ProjectStatus use:

   ```
   <property name="inactiveDesc"><null/></property>
   ```

2) The listvalues table, for the listID in the list table where listname = 'ProjectStatus", needs to have the identical set of ProjectStatus values defined, where the column valueKey="AUTOPSY", e.g. the first listValues record would be:

   ListID=NNN
   instance=lava
   scope=app-xyz
   ValueKey=AUTOPSY

ValueDesc=REFERRED
OrderID=1

And there would be a listValues record for each ProjectStatus value, i.e. “DEFERRED”, “UNDER_CONSIDERATION”, etc.

Note that unlike other dynamic lists, Enrollment Status ProjectStatus list is not a union of the values for a specified project (e.g. AUTOPSY) and the values for ValueKey=GENERAL; rather it is just a list of values for the specified project, and if the project does not customize the ProjectStatus values such that there is no list for the specified project, then it instead uses the “GENERAL” ProjectStatus values. This means that even if a project will only customize one ProjectStatus value and will use the default values for the rest, the listValues must still include all of the default ProjectStatus values in addition to the single customized value.

Note: if customizing the ProjectStatus values across all projects, i.e. the “GENERAL” listvalue records, then the bean definition in the “Autopsy” example above would be modified to replace “Autopsy” with “ANY” (to match all projects) and would define values for the *Desc properties, and the “GENERAL” records in the listValues table for the list “ProjectStatus” would have to be updated (to match those in the bean definition). Such updates should be appended to the lava-app-xyz-data.sql script by editing the database extract-app-xyz-data.sql script that creates it, e.g. appending to the end of extract-app-xyz-data.sql:

```
SELECT ‘UPDATE listvalues SET ValueDesc=’’CANCELLED’’ WHERE ValueKey=’’GENERAL’’ AND ValueDesc=’’DECLINED’’
AND ListID IN (SELECT ListID FROM list WHERE ListName=’’ProjectStatus’’ AND instance=’’lava’’ AND scope=’’crms’’);’;
```

In this case there would only be an update for each ProjectStatus value that differed from the default.

If not all of the project statuses are needed, then in both the bean definition and the listvalues table the *Desc value can be set to the empty string.

Defining Modules (Tabs) and Sections (Sub-tabs)

An application will generally define which module and sections it uses. Modules correspond to tabs in the user interface, while a module is subdivided into sections, which are links that appear under a module, like sub-tabs. The exception would be if an application used the exact same modules and sections as the lava-crms, in which case it would not have to define them.

Decorators

The tabs and sub-tabs are displayed within the “main” and “modal” decorators. Decorators are part of SiteMesh which is a Java web application framework used by LAVA that allows a clean separation of content from presentation. Decorators provide the look and feel of the application and provide a consistent look and feel across the various forms. The Sitemesh decorators are defined in lava-core under the WEB-INF directory in:

/WEB-INF/decorators/config/core-decorators.xml

Applications can define additional decorators in their own config files, such as:
WEB-INF/decorators/config/demo-decorators.xml
and the build.xml “merge-decorator-files” target (upon which the “deploy” target depends) merges the decorators from each Eclipse project into a single decorator in the deploy directory.

The types of decorators are:

- “main” and modal” decorators are the two different page level decorators, i.e. they decorate an entire web page. “modal” is used instead of “main” in cases where the user input is modal, i.e. the user should only be inputting data and not clicking any links on the page other than save and cancel, so in modal all other links are disabled. modal is usually in effect when editing an entity.
- the “panel” directory contains decorators for the left-hand action and reports panel
- decorators can be nested and the “component” directory contains the decorators that wrap the content, e.g. the entity and entity list decorators.

The decorator definitions map a decorator name, such as “main” with a jsp file, where the decorator jsp files are relative to “WEB-INF/decorators”, so “main” and “modal” are decorated by, respectively:

- lava-core/WEB-INF/decorators/main.jsp
- lava-core/WEB-INF/decorators/modal.jsp

Module Tabs

Both main.jsp and modal.jsp have a CSS style “tabBar” within which the tabs are output in HTML. The tabs are defined by the following HTML snippet:

```
<div id="tabBar">
    <c:set var="modulesURL"><spring:message code="${webappInstance}.modulesURL"
            text="navigation/tabBar/modules.jsp"/></c:set>
    <c:import url="/WEB-INF/jsp/${modulesURL}"/>
</div>
```

In lava-app-demo the tabs are defined in an i18n properties file with the following property key:
demo.modulesURL
The i18n properties file used by applications for strings related to modules and sections is the custom properties file, which for lava-app-demo is:
WEB-INF/i18n/source/app-demo-custom.properties

Looking in app-demo-custom.properties the value for demo.modulesURL is:
demo.modulesURL=/local/demo/navigation/tabBar/modules.jsp

which gives the location of the jsp file used to output the tabs in the user interface.

WEB-INF/jsp/local/demo/navigation/tabBar/modules.jsp contains:

```
<%@ include file="/WEB-INF/jsp/includes/include.jsp" %>
<ul>
    <tags:crmsNavTab module="home" text="Home"/>
    <tags:crmsNavTab module="people" text="People"/>
    <tags:crmsNavTab module="enrollment" text="Enrollment"/>
    <tags:crmsNavTab module="scheduling" text="Scheduling"/>
    <tags:crmsNavTab module="assessment" text="Assessment"/>
    <tags:crmsNavTab module="reporting" text="Reporting"/>
    <tags:ifHasRole roles="SYSTEM ADMIN">
        <tags:coreNavTab module="admin" text="Admin"/>
    </tags:ifHasRole>
```

```
</ul>
```

This shows that lava-app-demo has seven tabs, and that the “admin” module tab is only displayed if a user with the role “SYSTEM ADMIN” is logged in. The text argument passed for each tab is used as the label for the tab.

modal.jsp uses the same technique, so lava-app-demo resolves to:
demo.modalModulesURL=/local/demo/navigation/tabBar/modalModules.jsp
Note that the tabs are disabled because it is modal.

Section Sub-tabs

Defining sections is very similar to defining modules. The HTML snippet in the modal.jsp is:

```
<div id="tabBarSpacer">
	<div id="loginInfo">
		${pageContext.request.remoteUser}
	</div>
	<c:set var="modalSectionsURL"><spring:message code="${webappInstance}.modalSectionsURL"
		text="navigation/tabBar/modalSections.jsp"/></c:set>
	<c:import url="/WEB-INF/jsp/${modalSectionsURL}"/>
</div>
```

In app-demo-custom.properties the message key value pair is:
demo.modalSectionsURL=local/demo/navigation/tabBar/modalSections.jsp

The “people” module sections of modalSections.jsp look like:

```
<tags:crmsNavSection module="people" section="findPatient" textCode="section.findPatient" />
<tags:crmsNavSection module="people" section="patient" textCode="section.patient" />
<tags:crmsNavSection module="people" section="attachments" textCode="section.attachments" />
<tags:crmsNavSection module="people" section="caregiver" textCode="section.caregiver" />
<tags:crmsNavSection module="people" section="contactLog" textCode="section.contactLog" />
<tags:crmsNavSection module="people" section="contactInfo"textCode="section.contactInfo" />
<tags:crmsNavSection module="people" section="doctor" textCode="section.doctor" />
<tags:crmsNavSection module="people" section="task" textCode="section.task" lastSection="true"/>
```

One difference is that the labels for the section links are not supplied directly as text like they are for the modules. Instead, they are message keys where the keys are in the mvc properties file. For the default “people” module sections, the labels are in:
lava-crms/WEB-INF/i18n/source/crms-mvc.properties

note: the modules jsp files should be changed so that the labels are also in a properties file in keeping with the centralization of all text in properties file and the metadata table. This is useful for things like internationalization.

If a new section that did not exist in lava-core or lava-crms were added, the label would be added to the app mvc properties file, e.g. app-demo-mvc.properties.

Default Actions

When a user clicks on a module tab or a section sub-tab, the user is taken to a default view for the module or section. Every module and section must have a default. The defaults are defined as part of defining actions. When an action is defined that is a module or section default (or both), part of the action definition includes designating it as such.

In lava-core there are some predefined abstract action bean definitions created as a convenience for simplifying action definitions that are module and/or section defaults. These can be found in:
lava-core/WEB-INF/context/core/core-actions.xml

Within this file the following defines an action as both the module default and section default. It also happens to define an action as being a list in terms of the LAVA flow architecture, but that is besides the point here.

```
<bean id="coreModuleListAction" abstract="true" parent="coreListFlowAction">
    <property name="moduleDefault" value="true"/>
    <property name="sectionDefault" value="true"/>
</bean>
```

As an example, the definition of the action which is both the “admin” module default and the “auth” section default is:
lava-core/WEB-INF/context/core/core-auth.xml

```
<bean id="lava.core.admin.auth.authUsers" parent="coreModuleListAction"/>
```

The action shows a list of all of the LAVA users. Because it is the module default, this is the action that is performed when the user clicks on the “Admin” tab. And because it is also the section default this is the action that is performed when the user clicks on the “Authorization” sub-tab of the “Admin” tab.

Another convenience action definition in core-actions.xml is:

```
<bean id="coreSectionListAction" abstract="true" parent="coreListFlowAction">
    <property name="sectionDefault" value="true"/>
</bean>
```

An example using this bean definition is:
lava-core/WEB-INF/context/core/core-session.xml

```
<bean id="lava.core.admin.session.currentLavaSessions.lava"
    parent="coreSectionListAction"/>
```

The action displays a list of all current LAVA sessions. Because it is designated as the section default for the “admin.session” section, when the user clicks on the “Session Mgmt” sub-tab of the “Admin” tab, this is the action that is peformed.

In applications that are built from lava-crms there are different defaults based on the patient context and the project context. If there is a current patient context then a user is taken to the patient module default when they click on a module tab and a patient section default when they click on a section sub-tab. When there is not a current patient in context, then there is a project context (where the project context is All Projects if no project is selected in the project filter) and the user is taken to the project module default and project section default when clicking on a module tab and section sub-tab, respectively. It makes sense to show the current patient record when clicking on the Patient tab if there is a current patient, whereas it makes sense to show the Find Patient form or a list of all patients when there is not a current patient.

Just as in lava-core, there are some convenience actions defined that can be reference as the parent by action bean definitions that are designated as a module and/or section default.
lava-crms/WEB-INF/context/crms/crms-actions.xml defines

- crmsPatientSectionListAction section default when there is a current patient
- crmsPatientModuleListAction module/section default when there is a current patient
- crmsSectionListAction section default when there is not a current patient, i.e.project context section default
- crmsModuleListAction module and section default when there is not a current patient, i.e. project context module and section default

Usage examples of these can be found in the various lava-crms module context files under lava-crms/WEB-INF/context/crms, e.g.

- crms-people.xml
- crms-scheduling.xml
- crms-enrollment.xml
- crms-assessment.xml

Note that the reason the convenience actions in crms-actions.xml all happen to be list flow actions is because in most cases the default action for modules and sections is to display a list, e.g. the “Scheduling” tab module default is either a list of patient visits (the patient module default) or a list of all visits for the current project or all projects if no current project is selected in the project filter (this is the project module default).

Customizing Action and Reports Panel

The left navigation panel that is part of the decorator for all web pages that use the “main” decorator contains links for actions and reports that are contextually relevant the main content being displayed. These can be customized by redefining their location and using a different file to populate them.

The file used for this is the i18n custom properties file. For example, if lava-app-xyz were to customize the actions that are displayed for the “people” module, “patient” section to include a link to the list of diagnoses for the current patient, the file:
lava-app-xyz/WEB-INF/i18n/source/app-xyz-custom.properties
would contain a message key whose value is the content file to use in the action panel:
xyz.crms.people.patient.navActionsURL=local/xyz/crms/people/patient/actions.jsp

The decorator will always first look for the above instance specific message key when it is looking for the “people” module, “patient” section actions panel content. If such a file does not exist then the decorator will default to the following content file:
lava-crms/WEB-INF/jsp/crms/people/patient/actions.jsp

The content of the reports panel, which essentially shares the action panel located below the action links, would be customized in the same way, where the last part of the message key is “navReportsURL” instead of “navActionsURL”.

Skip Logic

LAVA has support for skip logic in views via the uitags JSTL tag library. The tag library descriptor documenting this library can be found in the LAVA CVS source code repository in: lava-core/WEB-INF/functions/uitags.tld. Currently LAVA is only using the “formGuide” tags. This tag library uses a javascript to implement tag functionality. That open source tag library has been modified for LAVA with the addition of new attributes and tags. There is an Eclipse project called “uitags” in the LAVA CVS repository with the source code used to build the LAVA version of uitags.

Following is a usage example from lava-crms
WEB-INF/jsp/crms/people/patient/patientContent.jsp

```
<!— do  not include skip logic when in ‘vw’ mode, i.e. readonly view →
<c:if test="${componentMode != 'vw'}">
        <ui:formGuide>
<!—when the deceased property is not 1=Yes (regexp [^1]) or is blank (regexp ^$) then disable the deathMonth, deathDay and
deathYear fields →
                <ui:observe elementIds="deceased" component="${component}" forValue="[^1]|^$"/>
                <ui:disable elementIds="deathMonth" component="${component}"/>
                <ui:disable elementIds="deathDay" component="${component}"/>
                <ui:disable elementIds="deathYear" component="${component}"/>
        </ui:formGuide>

<!—ignoreDoOnLoad means that this tag should not execute when the page is loading, because it is only intended for when a
user checks or unchecks the deidentified property checkbox.
The simulateEvents=”true” should be set on one of the formGuide tags on a page, typically the last, so that the formGuide tags are
checked when the page is loading →
        <ui:formGuide observeAndOr="or" ignoreDoOnLoad="true" simulateEvents="true">
                <!—the observeAndOr=”or” above indicates that if any of the observe tags are true then the action tag should be done.
                So if the deidentified property checkbox is checked, or if it is unchecked, then submit the form with the “reRender” event,
                which will result in the handler and view coordinating to change the patient record to or from deidentified. →
                <ui:observe elementIds="${component}_deidentified" forValue="1"/>
                <ui:observeForNull elementIds="${component}_deidentified"/>
                <ui:submitForm form="${component}" event="${component}__reRender"/>
        </ui:formGuide>
</c:if>
```

List Definition

A list in LAVA represents the domain of possible values that a property can assume. List definitions are configured in an XML configuration file and are referenced in the property metadata to then populate the input control for that property when it is displayed on a form. The list can also be used to translate a numeric property value into a string equivalent for display purposes.

List items are a name-value pair which facilitates having a label for display purposes which is associated with a numeric value which is persisted for analysis purposes. However, list items do not have to utilize the label / value structure. If the list item has a string value which suffices as the label, then the label part of the list item may be unused.

There are two general type of lists:

- “static” lists can be built when the application is loaded
- “dynamic” lists are created real-time when the list is needed via a query

While lists are defined in XML, the data that composes the list can either be in the XML, as part of the list definition, or in the database, where the list definition includes a query used to obtain the data. Much of the list data is stored in the “list” and “listvalues” tables in the database, but list data is not restricted to these tables.

In addition to the data, the other attributes of a list definition include:

- format is used to define aspects of how the list items are displayed, e.g. for a list with labels and corresponding numeric values the format “valueConcatLabel” results in a display of the numeric value concatenated with the label (separated by a ‘|’ char) which facilitates rapid data entry via numeric keys
- sort is used to designate how the list should be sorted. Lists can be sorted in the database or in LAVA and the sort attribute can designate which, and if in LAVA it designates whether the list should be sorted by label or value
- codes are specific to instrument lists where the domain of most instrument variables includes what are known as the standard error codes, while some instruments also include what are known as the skip error codes. See the instrument “LAVA Error Codes” section for more about codes.

Note: the full set of format and sort attribute values can be see in the lava-core class edu.ucsf.lava.core.list.model.BaseListConfig

While all properties have a “list” attribute in the property metadata the list is only used if the property will use a user input control that can accommodate a list. The control that is used is determined by a combination of things in the createField tag, such as the “style” metadata value. Note that a list style of “suggest” results in a dropdown control (known as an autocomplete box) that will display the specified list, but where the user selection is not restricted to the list such that the user can input a value that is not in the list, hence the list items are just a suggestion for the property value.

Property metadata includes a “listAttributes” attribute which is used to augment the list that the property uses. This facilitates two or more properties sharing the same list definition where the list needs to be augmented in some way for one or more of these properties.

The best way to learn the practical application of list definition is to look at the many definitions that already exist in the system. List definition is typically done with a set of building block definitions, where an application list definition uses one or a chain of parent references to inherit list attributes from other definitions. Lists definitions can be found in the following places in the LAVA source code respository:

- lava-core/WEB-INF/context/core/lists
- lava-crms/WEB-INF/context/crms/lists
- lava-crms-nacc/WEB-INF/context/crms/nacc-lists.xml
- lava-app-msc/WEB-INF/context/local/msc/msc-instruments.xml
- lava-crms-smd/WEB-INF/context/crms/crms-smd-devices.xml
- lava-crms-smd/WEB-INF/context/crms/crms-instruments.xml

Deidentified Patient Record

The Patient entity has a “deidentified” property. When this property is set (via a checkbox on the Edit Patient view) the patient lastName is set to the string “DE-IDENTIFIED’, other patient identifying properties are set to null, and the patient firstName property is turned into a subjectId property, which can be any ID that will identify the patient. To see more about how this is implemented see the Patient class and patientContent.jsp in lava-crms.

Customizing Entities and Actions

LAVA can be customized both in terms of properties (entities) and behavior (actions). New properties can be added to a new subclass of the customized entity (and to a new database table and Hibernate mapping, as well as a new view used to generate the web form). Behavior can be customized in many ways, such as setting a default value, performing validation, loading dynamic lists to populate the dropdown for a custom property, or performing other logic. Depending on what needs to be customized, the entity can be extended (subclassed), the entity handler can be extended or both.

There are two different ways to customize an entity action:

1) An “instance customization” customizes the entity action statically, i.e. the same extended entity and extended handler are used for the entity throughout the application instance.
2) A “dynamic customization” customizes the entity action dynamically at run-time based upon logic which determines which extended entity and which extended handler should be used in a particular situation, e.g. based on the current project in context, a specific project based extended entity and extended handler could be used.

   note: on Add, with dynamic customization there is no mechanism to include custom properties on the Add view because at the time the view backing object is instantiated it is not yet known what type of subclass instance will be persisted

Extending Entity

If an entity will be extended, a prototype configuration is required so that LAVA knows what type of instance to instantiate. The prototype is used when adding an extended entity, e.g. Add Visit or Add Enrollment Status. LAVA has a notion of context, where there is a patient context (the current patient) and project context (the project with which an entity is associated). The prototype configuration facilitates customizing an entity with a project context specific extension, e.g. the EnrollmentStatus entity class could be extended with EpicEnrollmentStatus for a project called “Epic”, with properties that are specific to enrollment in Epic (for example some properties for the screening required for a patient to enroll in this project, or a set of properties tracking a patient’s completion of the protocol for the project).

Prototype configuration applies to entities that have a project context, i.e. a “project” property. Thus, the patient entity does not use prototype configuration (patients can be enrolled in many projects so do not have a single project context). The Patient entity can be extended, but it is an “instance customization” where the same extended Patient subclass is used throughout the instance.

Extending Handler

If a handler will be extended, a LAVA action definition is the mechanism by which LAVA determines which handler should be used. There are two different ways of defining actions depending on whether the action is an “instance customization” or a “dynamic customization”. The specifics of each type of definition will be detailed in the examples which follow.

Note that under the current design there is no way to only customize the action handler and not customize the view. Meaning that whether “instance” or “dynamic” customization, the system will use a custom handler and expect a custom view whose location is determined by the customization mechanism. So if you are just customizing some handler behavior without requiring any changes to the view, you still have to have a custom view file, which is just a jsp which includes the content for the customized entity from the base project, e.g. lava-crms.

However, conversely, if you are just customizing the view (i.e. with additional properties for view/edit/save) but do not need any custom handling, then you can simply configure the FormAction for the custom action to reference the base class handler.

Local

The term “local” is used to reference properties and code that are local to the application, i.e. not a part of the lava-core or lava-crms or any other projects that the application uses besides its own. The “local” directory is used to locate files that implement the customization so that they are not deployed to the same directory as the files they are customizing in other scopes, and LAVA then utilizes this directory structure to determine where to find files when a particular action has been customized. “local” is generally used for customization of an *existing* action. If an app is defining new actions and entities which do not exist in other scopes, then files that implement them do not have to go under the “local” directories and can be put in the standard locations, based on what scope they belong to

The “local” prefix is used when configuring Spring beans that are specific to an instance. These bean definitions are then merged with lava-core or lava-crms bean definitions to compose all of the definitions of a given type. For example, “instrumentPrototypes” is used to configure the automatic creation of instruments when a Visit of a particular Project and Visit Type is created. “instrumentProtoypes” is the merger of “crmsInstrumentPrototypes” and “localInstrumentPrototypes” where any prototypes that are part of lava-crms are defined in the “crmsInstrumentPrototypes” bean (which should not be modified by an application developer) and any prototypes that are specific to the application instance would be defined in an application configuration file (e.g. local-instruments.xml) within the “localInstrumentPrototypes” bean.

Instance Customization

The mechanism used for an instance customization of an entity involves a custom action definition (for customizing the handler and view) and in some cases (for entities with a project context) a prototype configuration.

For example, an instance customization of the Visit entity would define this action:
<bean id="xyz.crms.scheduling.visit.visit" parent="crmsEntityFlowAction"/>

This definition has the following two effects:

1) The bean name of the Spring MVC FormAction that will be used to handle all events will be:

XyzVisitFormAction, rather than the base bean, VisitFormAction. This custom FormAction bean can then be configured with whatever custom handler(s) will be used to handle the events, e.g.
edu.ucsf.lava.local.xyz.crms.scheduling.controller.XyzVisitHandler

2) The name of the view file to display the entity will resolve to:
lava-app-xyz/WEB-INF/jsp/local/xyz/crms/scheduling/visit/visit.jsp

Even though the same entity class is always instantiated in instance customization a prototype configuration must be used. This is because the technique using the prototype configuration to instantiate the entity subclass is part of the base class handler, which is used when adding such an entity, regardless of whether it is an instance or dynamic customization. In this case, the prototype configuration would look like this:

lava-app-xyz/WEB-INF/context/local/xyz/xyx-local.xml
```
<bean id=”localVisitPrototypes” class=”java.util.LinkedHashMap”>
    <constructor-arg><map>
        <entry key=”ANY”>
            <bean class=”edu.ucsf.lava.local.xyz.crms.scheduling.model.XyzVisit”/>
        </entry>
```

The key=”ANY” indicates that this is an instance customization and that for *every* Add Visit an object of XyzVisit will be instantiated. The same configuration is supported for Add Enrollment Status using the bean “localEnrollmentStatusPrototypes”.

As mentioned above, prototype configurations are not used for instance customization of the Patient entity because Patient does not have a project context. In the case of Add Patient, the Patient entity subclass would be instantiated in AddPatientHandler where the class name is hard-coded, i.e. it does not come from a bean configuration.

Following are specific examples of instance customizations.

Patient Entity Instance Customization

The patient entity can be customized with additional properties, new/modified behavior or a combination of the two. The patient entity has separate actions for adding and for everything else. So depending upon the nature of your customization, you may have to customize one or both of the actions. If you are extending the Patient class to add properties, then you must create custom action definitions for both the patient and addPatient actions.

Add Patient has a separate action because adding a patient also involves enrolling the patient into a project (because every patient must be enrolled in at least one project). Because Add Patient is modifying two different entities – patient and enrollmentStatus – it uses a data transfer object (DTO) as its form backing object (aka command object). This DTO is the AddPatientCommand class and contains references to both the Patient and EnrollmentStatus classes.

This example will extend the patient entity with an additional property. The property will be for storing a medical record number and is named “mrn”.

xyz_patient database table

This table essentially subclasses the lava-crms patient table, extending the entity Patient with properties that are custom to the lava-app-xyz app.

```
CREATE TABLE xyz_patient (
        PIDN int NOT NULL,
        mrn varchar(15),
        PRIMARY KEY (PIDN),
        UNIQUE KEY mrn_unique (mrn),
        KEY fk_patient (PIDN),
        CONSTRAINT fk_patient FOREIGN KEY(PIDN) REFERENCES patient(PIDN)
                ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

XyzPatient.hbm.xml

```
/* This is the Hibernate mapping for the subclass of the Patient class. Notice that the Hibernate “joined-subclass” element is used.
*/
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <joined-subclass name="edu.ucsf.lava.local.xyz.crms.people.model.XyzPatient" table="xyz_patient"
        extends="edu.ucsf.lava.crms.people.model.Patient" select-before-update="true">
                <key column="PIDN"/>
                <property name="mrn" column="mrn" type="string" length="15"/>
        </joined-subclass>
</hibernate-mapping>
```

edu.ucsf.lava.local.xyz.crms.people.model.XyzPatient

```
/* This is the subclass of the Patient class that extends it with the “mrn” property for storing a medical record number. */
package edu.ucsf.lava.local.xyz.crms.people.model;

import edu.ucsf.lava.core.model.EntityBase;
import edu.ucsf.lava.core.model.EntityManager;
import edu.ucsf.lava.crms.people.model.Patient;

public class XyzPatient extends Patient {
        public static EntityManager MANAGER = new EntityBase.Manager(XyzPatient.class);
        private String mrn;

        public  String getMrn() {
                return mrn;
        }
 }
```

xyz-people.xml

```
<!—Custom Action Definitions →

<!-- define actions for instance customization of the Add Patient action and View/Edit/Delete Patient action. Each action definition
is identical to the action definition in lava-crms that is being extended, except for the instance portion is changed from the default
instance of "lava" to the application instance "xyz"

An entity action typically represents the CRUD actions: Add, View, Edit and Delete, although for the patient entity this is broken up
(as explained above) into the action id ending in "addPatient" for Add and the action id ending in "patient" for View, Edit and
Delete. -->
<bean id="xyz.crms.people.patient.addPatient" parent="crmsEntityFlowAction"/>
<bean id="xyz.crms.people.patient.patient" parent="crmsEntityFlowAction"/>


<!-- Form Action Definitions -->

<!-- the instance portion of the action definition above, "xyz", is prepended to the target portion of the action definition, "addPatient"
(with the first character capitalized) to derive the name of the FormAction bean which will be used by Spring MVC to handle events
for the action. So it is critical that the bean definition use the correct name.

In the LAVA architecture, the FormAction bean handling events for an action delegates the handling of those events to the handler
or handlers defined for the FormAction bean. So the XyzAddPatientHandler will handle all of the events of the Add Patient action,
such as "saveAdd", and "cancel". →

<bean id="xyzAddPatientFormAction" parent="crmsEntityComponentFormAction"><constructor-arg><list>
      <bean class="edu.ucsf.lava.local.xyz.crms.people.controller.XyzAddPatientHandler" parent="crmsHandler"/>
</list></constructor-arg></bean>

<bean id="xyzPatientFormAction" parent="crmsEntityComponentFormAction"><constructor-arg><list>
      <bean class="edu.ucsf.lava.local.xyz.crms.people.controller.XyzPatientHandler" parent="crmsHandler"/>
</list></constructor-arg></bean>
```

edu.ucsf.lava.local.xyz.crms.people.controller.XyzPatientHandler

```
/* This is the handler class specified above in the xyzPatientFormAction bean definition. The xyzPatientFormAction bean will
delegate the handling of events for the View Patient, Edit Patient and Delete Patient actions to this handler.

In this example the handler simply indicates the class which will be used as the Spring MVC form backing object for the
FormAction, XyzPatient. */

package edu.ucsf.lava.local.xyz.crms.people.controller;

import edu.ucsf.lava.crms.people.controller.PatientHandler;
import edu.ucsf.lava.local.xyz.crms.people.model.XyzPatient;

public class XyzPatientHandler extends edu.ucsf.lava.crms.people.controller.PatientHandler {

        public XyzPatientHandler() {
            super();
            this.setHandledEntity("patient", edu.ucsf.lava.local.xyz.crms.people.model.XyzPatient.class);
        }

}
```

edu.ucsf.lava.local.xyz.crms.people.controller.XyzAddPatientHandler

```
/* This is the handler class specified above in the xyzAddPatientFormAction bean definition. The xyzAddPatientFormAction bean
will  delegate the handling of events for the Add Patient action to this handler.
```

In this example the handler instantiates the patient subclass that will be used, XyzPatient. Note that the AddPatientCommand is used as the Spring MVC form backing object, and that the XyzPatient instance is set on this backing object. This is a special case where a DTO is needed (discussed above); in most cases the entity class itself is used as the form backing object. */

```
package edu.ucsf.lava.local.xyz.crms.people.controller;

import org.springframework.webflow.execution.RequestContext;
import edu.ucsf.lava.crms.people.model.AddPatientCommand;
import edu.ucsf.lava.local.xyz.crms.people.model.XyzPatient;

public class XyzAddPatientHandler extends edu.ucsf.lava.crms.people.controller.AddPatientHandler {
	protected Object initializeNewCommandInstance(RequestContext context, Object command) {
		AddPatientCommand apc = (AddPatientCommand) super.initializeNewCommandInstance(context, command);
apc.setPatient(new XyzPatient());
return apc;
	}
}
```

jsp/local/xyz/crms/people/patient/patient.jsp
Because the action is defined with the instance part of “xyz” replacing the default instance of “lava” the system will look for the view for the action under “WEB-INF/jsp/local/xyz/crms” instead of under the default location of “WEB-INF/jsp”. The rest of the file path for the view is the same for the entity that is being extended, “/people/patient/patient.jsp”.

One approach to creating a jsp which extends another jsp is to start from a copy of patient.jsp from lava-crms. If the customization involves property modifications, then modify the body to include a local content file, patientContent.jsp, so the local patient.jsp would contain:

```
<c:import url="/WEB-INF/jsp/local/xyz/crms/people/patient/patientContent.jsp">
	<c:param name="component">${component}</c:param>
</c:import>
```

Then start the local patientContent.jsp with a copy of the lava-crms patientContent.jsp and modify it accordingly, adding properties, removing properties, rearranging properties, etc.
For example, add the following to the desired position in patientContent.jsp:

```
<tags:createField property=”mrn” component=”${component}”/>
```

Note that the local patientContent.jsp could import the lava-crms patientContent.jsp and then append new properties within a new section after the import. However, if the layout is such that this will not give the desired view, then copy the contents of the lava-crms patientContent.jsp to the local patientContent.jsp and make modifications.

Otherwise, if the customization does not involve any modifications to properties, the body can be left as is, including the lava-crms patientContent.jsp file:

```
<c:import url="/WEB-INF/jsp/crms/people/patient/patientContent.jsp">
	<c:param name="component">${component}</c:param>
</c:import>
```

jsp/local/xyz/crms/people/patient/addPatient.jsp
The addPatient.jsp would be customized in the same manner as patient.jsp

Visit and EnrollmentStatus Entity Instance Customization

Instance customization of Visit and EnrollmentStatus is virtually identical. One difference is that Add Enrollment Status has a separate handler, AddEnrollmentStatusHandler, for adding, and an EnrollmentStatusHandler for viewing, editing and deleting, while Visit has just a single VisitHandler for all CRUD actions. This has to do with the nature of the entities; EnrollmentStatus has a short form "quick add" because EnrollmentStatus usually undergoes a series of edits over time where the full EnrollmentStatus form is used, whereas the Visit form may be filled out completely at the time it is added, so the full Visit form is used for add.

In terms of instance customization this does not make much of a difference. It simply means that if Visit is customized, then the customizations will be seen on Add Visit as well as View/Edit/Delete, whereas if EnrollmentStatus is customized the customizations will not be seen on Add; only on View / Edit and Delete. It is certainly possible to customize AddEnrollmentStatus in the same way as Visit and EnrollmentStatus if one wanted to see customizations on Add Enrollment Status.

For an instance customization of EnrollmentStatus, follow the same steps as below for an instance customization of Visit. If you are also customizing AddEnrollmentStatus you would not have a constructor with a setHandledEntity method call because AddEnrollmentStatus uses a AddPatientCommand object as a DTO for its command object (form backing); instead the class used to instantiate is determined in the base class AddEnrollmentStatusHandler initializeNewCommandInstance method by calling getEnrollmentStatusPrototype, so you just need the localEnrollmentStatusPrototypes bean configuration with the "ANY" key.

Futhermore, VistHandler, EnrollmentStatusHandler and AddEnrollmentStatusHandler need not be subclassed if there is no custom handling needed. If the customization just involves additional properties for viewing, editing and persisting then the FormAction definitions can just reference the base class handlers. In the example below a subclass handler is used to set custom default values.

Note that EnrollmentStatus can also be customized in terms of its statuses, which can be done whether or not there is an instance customization of EnrollmentStatus. But if there is an instance customization of EnrollmentStatus then customizing statuses is done in conjunction with it, in the prototype configuration. See the "Customizing Enrollment Statuses" section.

Following is an example of both extending the visit entity with new properties and setting custom defaults for some of the lava-crms Visit properties.

xyz_visit database table

This table essentially subclasses the lava-crms visit table, extending the Visit entity with properties that are custom to the lava-app-xyz app.

```
CREATE TABLE xyz_visit (
	VID int NOT NULL,
	gilenya smallint DEFAULT NULL,
	gilenya_b smallint DEFAULT NULL,
	gilenya_f smallint DEFAULT NULL,
	vigabatrin smallint DEFAULT NULL,
	PRIMARY KEY (VID),
	KEY fk_visit (VID),
	CONSTRAINT fk_visit FOREIGN KEY(VID) REFERENCES patient(VID)
```

```
        ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

WEB-INF/hibernate/local/xyz/crms/scheduling/XyzVisit.hbm.xml

/* This is the Hibernate mapping for the subclass of the lava-crms Visit class. Notice that the Hibernate “joined-subclass” element is used. */

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <joined-subclass name="edu.ucsf.lava.local.xyz.crms.scheduling.model.XyzVisit" table="xyz_visit"
    extends="edu.ucsf.lava.crms.scheduling.model.Visit" select-before-update="true">
        <key column="VID"/>
        <property name="gilenya" column="gilyenya" type="short”/>
        <property name="gilenyaBaseline" column="gilenya_b" type="short”/>
        <property name="gilenyaFollowup" column="gilenya_f" type="short”/>
        <property name="vigabatrin" column="vigabatrin" type="short”/>
    </joined-subclass>
</hibernate-mapping>
```

edu.ucsf.lava.local.xyz.crms.scheduling.model.XyzVisit

/* This is the subclass of the Visit class that extends it with custom properties. */

```
package edu.ucsf.lava.local.xyz.crms.scheduling.model;

import edu.ucsf.lava.crms.scheduling.model.Visit;

public class XyzVisit extends Visit {
private Short gilenya;
private Short gilenyaBaseline;
private Short gilenyaFollowup;
private Short vigabatrin;

public Short getGilenya() {
    return gilenya;
}
public void setGilenya(Short gilenya) {
    this.gilenya = gilenya;
}
public Short getGilenyaBaseline() {
    return gilenyaBaseline;
}
public void setGilenyaBaseline(Short gilenyaBaseline) {
    this.gilenyaBaseline = gilenyaBaseline;
}
public Short getGilenyaFollowup() {
    return gilenyaFollowup;
}
public void setGilenyaFollowup(Short gilenyaFollowup) {
    this.gilenyaFollowup = gilenyaFollowup;
}
public Short getVigabatrin() {
    return vigabatrin;
}
public void setVigabatrin(Short vigabatrin) {
    this.vigabatrin = vigabatrin;
}
}
```

xyz-visit.xml

```
<!—Custom Action Definitions →
```

```
<!—define action for instance customization of the Visit entity. Each action definition is identical to the action definition in lava-crms that is being extended, except for the instance portion is changed from the default instance of “lava” to the application instance “xyz” -->
<bean id="xyz.crms.scheduling.visit.visit" parent="crmsEntityFlowAction"/>

<!-- Form Action definitions -->

<!-- the instance portion of the action definition above, “xyz”, is prepended to the target portion of the action definition, “visit” (with the first character capitalized) to derive the name of the FormAction bean which will be used by Spring MVC to handle events for the action. So it is critical that the bean definition use the correct name.

In the LAVA architecture, the FormAction bean handling events for an action delegates the handling of those events to the handler or handlers defined for the FormAction bean. So the XyzVisitHandler will handle all of the events for Visit entity CRUD such as “save”, and “cancel”. Note that more than one handler can be specified. In this example, the PatientVisitsHandler also handles events, because on Add Visit the user is shown a list of the already existing visits for the patient. The FormAction iterates thru all specified handlers until it finds one that will handle a given event (all handlers have a method that takes an event argument and returns whether or not the handler handles that event) →

<bean id="xyzVisitFormAction" parent="crmsEntityComponentFormAction"><constructor-arg><list>
	<bean class="edu.ucsf.lava.local.xyz.crms.scheduling.controller.XyzVisitHandler" parent="crmsHandler"/>
	<bean class="edu.ucsf.lava.crms.scheduling.controller.PatientVisitsHandler" parent="crmsHandler"/>
</list></constructor-arg></bean>
```

edu.ucsf.lava.local.xyz.crms.scheduling.controller.XyzVisitHandler

```
/* This is the handler class specified above in the xyzVisitFormAction bean definition. The xyzVisitFormAction bean will  delegate the handling of events for the Add / View / Edit / Delete Visit to this handler.

In this example the handler indicates the class which will be used as the Spring MVC form backing object for the FormAction, XyzVisit,  */

package edu.ucsf.lava.local.xyz.crms.scheduling.controller;

import…

public class XyzVisitHandler extends edu.ucsf.lava.crms.scheduling.controller.VisitHandler {

	public XyzVisitHandler() {
		super();
		this.setHandledEntity("visit", XyzVisit.class);
	}

	protected Object initializeNewCommandInstance(RequestContext context, Object command)  {
		super.initializeNewCommandInstance(context, command);
		// set defaults
		Visit v = (Visit) command;
		v.setVisitLocation("ACC-8");
		v.setVisitWith("Jones, N");
		v.setVisitDate(new Date());
		v.setVisitStatus("COMPLETE");
		return command;
	}

	public Map addReferenceData(RequestContext context, Object command, BindingResult errors, Map model) {
		Visit v = (Visit)((ComponentCommand)command).getComponents().get(getDefaultObjectName());
		// visitWith default is tricky because the uitags reRender on projName change sets visitWith to blank .
		// so that a visitWith value specific to one projName does not show up for another, have to
		// reset the default for visitWith here
		if (v.getVisitWith() == null) {
			v.setVisitWith("Jones, N");
		}
		return super.addReferenceData(context, command, errors, model);
	}
}
```

localVisitPrototypes

```
<!—this prototype configuration specifies that for any (“ANY”) project to which a Visit is assigned, an object of type XyzVisit should
be instantiated. This lookup is done on Add Visit in the lava-crms VisitHandler doSaveAdd method based upon what the user
chose for the Visit projName property. →
<bean id="localVisitPrototypes" class="java.util.LinkedHashMap">
	<constructor-arg><map>
		 <!-- customization so XyzVisit is created for all projects -->
		<entry key="ANY">
			<bean class="edu.ucsf.lava.local.xyz.crms.scheduling.model.XyzVisit"/>
		</entry>
	</map></constructor-arg>
</bean>
```

jsp/local/xyz/crms/scheduling/visit/visit.jsp

```
<%-- The technique used here is to include the lava-crms jsp/crms/scheduling/visit/visitContent.jsp and then to output the
extended Visit properties in a following section --%>

<%@ include file="/WEB-INF/jsp/includes/include.jsp" %>

<c:set var="component" value="visit"/>

<c:set var="modeString" value="${component}_mode"/>
<c:set var="componentMode" value="${requestScope[modeString]}"/>

<c:set var="viewString" value="${component}_view"/>
<c:set var="componentView" value="${requestScope[viewString]}"/>
<c:if test="${componentView == 'add'}">
	<c:set var="componentView" value="addMany"/>
</c:if>

<page:applyDecorator name="component.content">
	<page:param name="component">${component}</page:param>
	<page:param name="focusField">${flowEvent == 'visit__reRender' ? 'visitType' : 'projName'}</page:param>
	<page:param name="pageHeadingArgs">${currentPatient.fullNameNoSuffix},${currentVisit.visitDescrip}</page:param>

	<page:applyDecorator name="component.entity.content">
		<page:param name="component">${component}</page:param>
		<page:param name="componentView">${componentView}</page:param>
		<page:param name="locked">${command.components['visit'].locked}</page:param>

		<c:import url="/WEB-INF/jsp/crms/scheduling/visit/visitContent.jsp">
			<c:param name="component">${component}</c:param>
			<c:param name="componentMode">${componentMode}</c:param>
			<c:param name="componentView">${componentView}</c:param>
		</c:import>

		<page:applyDecorator name="component.entity.section">
			<page:param name="sectionId">clinic</page:param>
			<page:param name="sectionNameKey">visit.subproject.section</page:param>
			<tags:createField property="gilenya" component="${component}" labelAlignment="right checkboxRight"/>
			<tags:createField property="gilenyaBaseline" component="${component}" labelAlignment="right checkboxRight"/>
			<tags:createField property="gilenyaFollowup" component="${component}" labelAlignment="right checkboxRight"/>
			<tags:createField property="vigabatrin" component="${component}" labelAlignment="right checkboxRight"/>
		</page:applyDecorator>
	</page:applyDecorator>

<%-- the below should be refactored into another include from lava-crms visit.jsp instead of duplicating it because now if the lava-
crms visit.jsp changes then this code much be changed as well --%>

	<%-- if adding visit, show the list of other visits for the patient associated with primary visit
	    note: add visit is not accessible unless there is a patient in context, so that is guaranteed --%>
	<c:if test="${componentView == 'addMany'}">
		<c:set var="component" value="patientVisits"/>

		<page:applyDecorator name="component.list.content">
			<page:param name="component">${component}</page:param>
			<page:param name="pageName">visit</page:param>
			<page:param name="listTitle">Visits associated with
```

```
                <tags:componentProperty component="visit" property="patient" property2="fullName"/>
            </page:param>
            <page:param name="contextualInfo"> </page:param>

            <c:import url="/WEB-NF/jsp/crms/scheduling/visit/patientVisitsContent.jsp">
                <c:param name="component">${component}</c:param>
            </c:import>
        </page:applyDecorator>
    </c:if>

</page:applyDecorator>
```

Entity Dynamic Customization

Dynamic customization of an entity allows the handler and view used for the entity actions to be determined at runtime. For example, the EnrollmentStatus entity could be handled and displayed differently depending upon the project in which subjects are enrolled. There would be one handler for each project that needed such customization, and each handler would implement logic which determines whether or not the handler should be the handler that is used for a particular EnrollmentStatus action. The configuration essentially creates a chain of these handlers, giving each of them a shot at handling an action. If none of the handlers should handle the action, then the default EnrollmentStatus handler will handle it.

Any type of entity can be customized dynamically, and the logic used to determine whether a handler should handle an action can be anything, e.g. it can be based on any property or properties in the entity or associated entities.

The mechanism used for a dynamic customization of an entity involves a custom action definition and a custom handler that includes the logic to determine whether the handler should handle the entity action.

For example, given the projects “Epic” and “Riluzole” where both projects required custom handling, there would be the following action definitions:

```
<bean id="xyz.crms.enrollment.status.epicEnrollmentStatus" parent="crmsEntityFlowAction">
    <property name="customizedFlow"><value>lava.crms.enrollment.status.enrollmentStatus</value></property>
</bean>
<bean id="xyz.crms.enrollment.status.riluzoleEnrollmentStatus" parent="crmsEntityFlowAction">
    <property name="customizedFlow"><value>lava.crms.enrollment.status.enrollmentStatus</value></property>
</bean>
```

These definitions have the following effect:

- defining the “customizedFlow” property is how an entity inserts its handler into the chain of handlers that may handle the actions of the given entity type. The “customizedFlow” property must identify the base class handler that is being customized.
- on an EnrollmentStatus event the flow structure will dynamically iterate thru each FormAction bean in the chain, querying the handler as to whether it should handle the event

- if a given handler signals that it will handle the event then it will be used as the event handler and its corresponding view will be used: lava-app-xyz/WEB-INF/jsp/local/xyz/crms/enrollment/status/epicEnrollmentStatus.jsp

There are actually handler two methods involved in dynamic customization. These are both called for an action that has been configured for dynamic customization, i.e. the custom handler's implementation of these is called.

- preSetupFlowDirector exists as a hook in the LAVA flow infrastructure for global transition events, e.g. if the system or part of the system were down for maintenance it could be used to transition the user to a system maintenance web page. But for the purposes of dynamic customization it should return SUCCESS so that flow control proceeds normally. The default implementation in LavaComponentHandler does just this, so custom handlers need not implement it.
- postSetupFlowDirector is called after standard flow setup to determine whether the handler should handle the flow. The handler should return CONTINUE if it is to handler the flow. Otherwise it should return UNHANDLED, in which case the flow infrastructure will call preSetupFlowDirector .. postSetupFlowDirector for the next handler in the chain, or if there are no more custom handlers, flow control will transition back to use the default entity action handler. postSetupFlowDirector has a default implementation in LavaComponentHandler that returns CONTINUE so that default actions will always signal that they are handling the event (so if there are no custom handlers in the chain there is still a check to see if the default handler should handler the event, which of course it should).

Note that in between preSetupFlowDirector and postSetupFlowDirector, the standard flow setup methods are called, which include loading the entity.

Only one FormAction and its handlers will handle a given event. When a custom handler signals that it will handle the event, after the event is handled, no other handlers in the chain will be called and given a chance to handle the event.

Following is an implementation of preSetupFlowDirector that determines whether a handler should handle the action based on whether the Enrollment Status project is “Epic”. If the projName property is “Epic” then it return CONTINUE to signal to the flow infrastructure that it will handle the event.

```
public Event postSetupFlowDirector(RequestContext context, Object command) throws Exception {
	String flowMode = ActionUtils.getFlowMode(context.getActiveFlow().getId());
	if (flowMode.equals("add")) {
		return new Event(this,this.UNHANDLED_FLOW_EVENT_ID);
	}
	EnrollmentStatus enrollmentStatus
		= (EnrollmentStatus) ((ComponentCommand)command).getComponents().get(getDefaultObjectName());
	if (enrollmentStatus != null && enrollmentStatus.getProjName().toLowerCase().startsWith("epic")) {
		return new Event(this,this.CONTINUE_FLOW_EVENT_ID);
	}
	return new Event(this,this.UNHANDLED_FLOW_EVENT_ID);
}
```

Note that with dynamic customization that is based on properties of the entity the add flow is not customized (postSetupFlowDirector always returns UNHANDLED) because for the add action, the entity does not exist yet, so a decision cannot be made as to which handler should handle the entity and which model subclass should be instantiated. The implication is that custom properties can not appear on the add view and the user must view/edit subsequent to adding to see the custom properties. It is technically possible to dynamically customize an add flow if there is such a use case.

Visit and EnrollmentStatus Entity Dynamic Customization

Dynamic customization of Visit and EnrollmentStatus is conceptually identical. Following is an example of dynamically customizing the EnrollmentStatus entity based on the project (projName property). The “Epic” project requires an additional custom property, “forAnalysis”. If the EnrollmentStatus is for the “Epic” project then the custom EpicEnrollmentStatus model class should be used as the form backing object and for persistence, and the custom epicEnrollmentStatus.jsp view should be used for the entity form.

For dynamic customization of the Visit entity follow the same general steps.

epic_enrollmentstatus table

This table essentially subclasses the lava-crms visit table, extending the EnrollmentStatus entity with a property that is custom to both the lava-app-xyz app and the “Epic” project.

```
CREATE  TABLE epic_enrollmentstatus (
	EnrollStatID INT NOT NULL ,
	for_analysis BOOLEAN NULL DEFAULT NULL,
	PRIMARY KEY (EnrollStatID) ,
	KEY fk_enrollmentStatus (EnrollStatID),
	CONSTRAINT fk_enrollmentStatus FOREIGN KEY(EnrollStatID) REFERENCES enrollmentstatus(EnrollStatID)
		ON DELETE NO ACTION ON UPDATE NO ACTION
)
ENGINE = InnoDB; DEFAULT CHARSET latin1;
```

WEB-INF/hibernate/local/xyz/crms/enrollment/EpicEnrollmentStatus.hbm.xml

/* This is the Hibernate mapping for the subclass of the lava-crms EnrollmentStatus class. Notice that the Hibernate “joined-subclass” element is used. */

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
	<joined-subclass name="edu.ucsf.lava.local.xyz.crms.enrollment.model.EpicEnrollmentStatus" table="epic_enrollmentstatus"
	extends="edu.ucsf.lava.crms.enrollment.model.EnrollmentStatus" select-before-update="true">
		<key column="EnrollStatID"/>
		<property name="forAnalysis" column="for_analysis" type="boolean”/>
	</joined-subclass>
</hibernate-mapping>
```

edu.ucsf.lava.local.xyz.crms.enrollment.model.EpicEnrollmentStatus

/* This is the subclass of the EnrollmentStatus class that extends it for the “Epic” project with a custom property. */

```
package edu.ucsf.lava.local.xyz.crms.enrollment.model;

public class EpicEnrollmentStatus extends EnrollmentStatus {
	private Boolean forAnalysis;

	public EpicEnrollmentStatus() {
		super();
	}

	public Boolean getForAnalysis() {
		return forAnalysis;
	}

	public void setForAnalysis(Boolean forAnalysis) {
		this.forAnalysis = forAnalysis;
	}
}
```

xyz-enrollmentstatus.xml

```
<!—Custom Action Configuration ->

<!—define action for dynamic customization of the EnrollmentStatus entity. The action definition is identical to the action definition
in lava-crms that is being customized, except for two changes:
	1. the instance (first part) is changed from the default instance of “lava” to the application instance “xyz”
	2. the target portion (last part) is changed from the default “enrollmentStatus” to the custom “epicEnrollmentStatus"
-->
<bean id="xyz.crms.enrollment.status.epicEnrollmentStatus" parent="crmsEntityFlowAction">
	<property name="customizedFlow"><value>lava.crms.enrollment.status.enrollmentStatus</value></property>
</bean>

<!-- Form Action definitions -->

<!-- the instance portion of the action definition above, “xyz”, is prepended to the target portion of the action definition,
“epicEnrollmentStatus” (with the first character capitalized) to derive the name of the FormAction bean which will be used by
Spring MVC to handle events for the action. So it is critical that the bean definition use the correct name.

In the LAVA architecture, the FormAction bean handling events for an action delegates the handling of those events to the handler
or handlers defined for the FormAction bean. So in the case where XyzEpicEnrollmentStatusHandler dynamically signals that it
should handle events in its postSetupFlowDirector method, the xyzEpicEnrollmentStatusFormAction will be used by the LAVA flow
infrastructure will be used for handling the event and will delegate that handling to XyzEpicEnrollmentStatusHandler →

<bean id="xyzEpicEnrollmentStatusFormAction" parent="crmsEntityComponentFormAction">
	<constructor-arg><list>
		<bean class="edu.ucsf.lava.local.xyz.crms.enrollment.controller.EpicEnrollmentStatusHandler" parent="crmsHandler"/>
	</list></constructor-arg>
</bean>
```

EpicEnrollmentStatusHandler

```
/* This is the handler class specified above in the xyzEpicEnrollmentStatusFormAction bean definition. This bean will  delegate the
handling of events for Add / View / Edit / Delete Visit to this handler, which will determine whether or not it should handle the
event, based on what is returned from the postSetupFlowDirector method. */

package edu.ucsf.lava.local.xyz.crms.enrollment.controller;

public class EpicEnrollmentStatusHandler extends EnrollmentStatusHandler {

public EpicEnrollmentStatusHandler() {
	super();
	/* specify the subclass that will be used for the Spring MVC form backing object */
	this.setHandledEntity("enrollmentStatus", EpicEnrollmentStatus.class);
	/* this is needed to get around ClassCastException when dealing with multiple subclasses of the base class */
	this.setDefaultObjectBaseClass(EnrollmentStatus.class);
}

public Event postSetupFlowDirector(RequestContext context, Object command) throws Exception {
        String flowMode = ActionUtils.getFlowMode(context.getActiveFlow().getId());
        if (flowMode.equals("add")) {
                return new Event(this,this.UNHANDLED_FLOW_EVENT_ID);
        }
        EnrollmentStatus enrollmentStatus
                = (EnrollmentStatus) ((ComponentCommand)command).getComponents().get(getDefaultObjectName());
        /* this is where the logic goes which determines whether or not this handler should handle the event */
        if (enrollmentStatus != null && enrollmentStatus.getProjName().toLowerCase().startsWith("epic")) {
                return new Event(this,this.CONTINUE_FLOW_EVENT_ID);
        }
        return new Event(this,this.UNHANDLED_FLOW_EVENT_ID);
}
```

jsp/local/xyz/crms/enrollment/status/epicEnrollmentStatus.jsp

```
<%-- because the instance part of the action "xyz.crms.enrollment.status.epicEnrollmentStatus" is instance specific (and not  the
default “lava” instance), the view resolves to a location under the “WEB-INF/jsp/local” directory, followed by the instance, “xyz”.
Following that is the same scope/module/section directory path used in lava-crms, i.e. “crms/enrollment/status”. The file itself is
then named after the target part of the action “epicEnrollmentStatus” with .jsp appended. --%>



<%@ include file="/WEB-INF/jsp/includes/include.jsp" %>

<c:set var="component" value="enrollmentStatus"/>

<page:applyDecorator name="component.content">
        <page:param name="component">${component}</page:param>
        <page:param name="quicklinks">controls,statusHistory</page:param>
        <page:param name="pageHeadingArgs"><tags:componentProperty component="${component}"
                property="projName"/>,${currentPatient.fullNameNoSuffix}</page:param>

        <page:applyDecorator name="component.entity.content">
                <page:param name="component">${component}</page:param>
                <page:param name="locked">${command.components['enrollmentStatus'].locked}</page:param>

                        <c:import url="/WEB-INF/jsp/crms/enrollment/status/enrollmentStatusContent.jsp">
                                <c:param name="component">${component}</c:param>
                        </c:import>

                        <c:import url="/WEB-INF/jsp/crms/enrollment/status/epicEnrollmentStatusContent.jsp">
                                <c:param name="component">${component}</c:param>
                        </c:import>

                        <c:import url="/WEB-INF/jsp/crms/enrollment/status/enrollmentStatusHistoryContent.jsp">
                                <c:param name="component">${component}</c:param>
                        </c:import>
        </page:applyDecorator>
</page:applyDecorator>
```

Note that while the use of a prototype configuration is required to instantiate a custom subclass of an entity class, this is not part of the dynamic customization configuration described above. This is done in the same way for both instance and dynamic customization.

For example, along with the dynamic customization of the EnrollmentStatus entity for the “Epic” project, the following prototype configuration must exist so that in the default Add Enrollment Status an EpicEnrollmentStatus is instantiated.

lava-app-xyz/WEB-INF/context/local/xyz/xyx-local.xml

```
<bean id=”localEnrollmentStatusPrototypes” class=”java.util.LinkedHashMap”>
        <constructor-arg><map>
                <entry key=”Epic”>
                        <bean class=”edu.ucsf.lava.local.xyz.crms.enrollment.model.EpicEnrollmentStatus”/>
                </entry>
```

This will ensure that Add Enrollment Status will instantiate an object of the EpicEnrollmentStatus class when a patient is enrolled in the Epic project (see AddEnrollmentStatus doReRender method). The enrollment status values of this custom class can also be customized in this same prototype configuration. Refer to the “Customizing Enrollment Statuses” section.

Entity Prototype Default Values

Entity prototype configuration can be used for more than just specifying which entity subclass should be instantiated when a given type of entity is added. They can be used to set initialize values on the entity object when it is instantiated.

Here is an example of the localVisitPrototypes from lava-app-demo in the file WEB-INF/context/local/demo/demo-scheduling.xml

When the user adds a Visit where projName is “UDS” the Visit subclass UdsVisit will be instantiated and the visitType that the user specifies sets the packet property of this UdsVisit object accordingly.

```
<bean id="localVisitPrototypes" class="java.util.LinkedHashMap">
    <constructor-arg><map>
        <entry key="UDS~Initial Assessment">
            <bean class="edu.ucsf.lava.crms.scheduling.model.UdsVisit">
                <property name="packet" value="I"/>
            </bean>
        </entry>
        <entry key="UDS~Follow Up Assessment">
            <bean class="edu.ucsf.lava.crms.scheduling.model.UdsVisit">
                <property name="packet" value="F"/>
            </bean>
        </entry>
        <entry key="UDS~Telephone Follow Up">
            <bean class="edu.ucsf.lava.crms.scheduling.model.UdsVisit">
                <property name="packet" value="T"/>
            </bean>
        </entry>
    </map></constructor-arg>
</bean>
```

Possible visitType values are “Initial Assessment”, “Follow Up Assessment”, or “Telephone Follow Up”.

Instruments

LAVA uses the term instrument for clinical research management assessments. These assessments may be longitudinal in nature, and every instrument has an associated Visit entity (as well as Patient entity). LAVA instruments can range from fairly simple, such as a straightforward single record web form, to complex instruments that may have a single record and one or more lists of detail records, or instruments that require file loading rather than data entry. This section will go thru the steps of creating a straightforward instrument.

Instrument Creation

The process of creating an instrument has been automated to a certain extent. “Code generation” techniques are employed to assist in generating the “create table” statement for the underlying database table, generating the files required by the instrument: Hibernate mapping file, Java class file, jsp view file, and populating the viewproperty table with metadata for each instrument field. The underlying database table is created and metadata is inserted. Additionally, there are some configuration steps involved.

The code generation requires a suite of stored procedures. As of this writing there is only a suite for the MySQL database. Make sure the database has these stored procedures before starting instrument creation. The stored procedures all begin with “util_”. If the database does not have these, obtain the latest version from CVS:
lava-core/database/core/development/util/core-util-procs.sql

Run the script, which may require that you are the root database user.

instrTypeEncoded
Before proceeding, the term “instrTypeEncoded” should be clarified, as it is used frequently in the following documentation. The name chosen for the instrument is stored in the “instrType” property of the instrument class. This name can have a maximum of 25 characters, and can contain spaces and non-alphanumeric characters. Because there is a need to specify the instrument type in situations where spaces and non-alphanumeric characters are not allowed, such as in URLs and Spring bean names, an instrument has an alternate naming for these internal uses, and this is known as “instrTypeEncoded”, where instrTypeEncoded is generated by removing all non-alphanumeric characters (including spaces) and converting to all lowercase.
e.g. given an instrument name of “SF-36”, instrTypeEncoded is “sf36”

Instrument Creation Steps

1) Begin with the datadictionary_template.xlxs file in /lava-crms/development/instruments/. You could copy this file to a working directory for your new instrument and rename it, e.g. datadictionary_sf36.xlsx. All data filled out in this form will be inserted into the datadictionary table. The purpose of filling this out completely is two-fold:
   a) Allows more streamlined automation of code to create the instrument, since most data and metadata needed for instrument creation are contained in this form.

b) The datadictionary table forms the basis for the data dictionary of all LAVA variables. A data dictionary can be generated from the table and made available for LAVA users. A LAVA Query data object could be created for the data dictionary so that users could always have access to the most up-to-date variable definitions.

   Here are explanations of all the fields you should fill out:

scope	used to distinguish variables at core, crms and app scope, e.g. ‘app-xyz’
entity	instrTypeEncoded, e.g. ‘moca’
prop_order	(optional) order of the property. not used on web forms, but may be useful for other purposes, like report generation
prop_name	name of the property; this is the Java class property name you will be using
prop_description	a full description of the property; usually the full text of the question
data_values	if there is a defined domain, the allowable values for this property
data_calculation	if the property will be automatically calculated, write in the pseudocode
required	is this a required field or not; 1=Yes; 0=No
db_table	database table name; can be whatever you want but should reflect the instrument name. For MySQL, should be all lowercase
db_column	the column names in the database table; db_column is typically the same as prop_name, but you may prefer a different naming convention; for example, db_column can be all lowercase with words separated by ‘_’ and prop_name can be titleCase with no underscores
db_order	order of the db_columns, used when generating CREATE TABLE
db_datatype	the column data type
db_datalength	the length of the column data type, if applicable
db_nullable	can column be null; 1=Yes; 0=No
db_default	default column value, if applicable

As an alternative to Steps 2) thru 10) below you can instead follow the instructions in “Streamlined Creation Scripts” following Step 10) for a more automated instrument creation process. You may want to follow Steps 2) thru 10) for your first instrument or two so that you gain a thorough understanding of the steps involved and then used the streamlined creation scripts thereafter.

2) After you have completely filled out all fields, column C of the datadictionary xlxs has all the INSERT statements needed to insert your data into the datadictionary table. Copy & Paste these INSERT statements into your SQL editor and execute on your development database.

3) There are two utility stored procedures that will help you generate the CREATE TABLE syntax for your instrument:
   a) util_createtable, passing in the following parameters: entity, scope

b) util_tablekeysadd, passing in: entity, scope

Using the syntax generated from these two stored procedures, execute it on your dev database.

4) The util_addTableToHibernateProperty utility stored procedure will read the schema definition for the new instrument table and populate the hibernateproperty table with data for each variable that can be used to generate the heart of the instrument's Hibernate mapping file and Java class file.

   Execute util_addTableToHibernateProperty, passing in:
   a) instrument table name
   b) entity name (instrTypeEncoded)
   c) scope (used to distinguish variables at core, crms and app scope)

   e.g. call util_AddTableToHibernateProperty('sf_36', 'sf36', 'app-smd')

5) The util_addTableToMetaData utility stored procedure will read the schema definition for the new instrument table and populate the viewproperty table with metadata for each property, used in displaying the property on a view. The viewproperty metadata is also used in the instrument creation process to code generate the required result fields code for the instrument's Java class file and the createField tags for the instrument's jsp file.

   Execute util_addTableToMetaData, passing in:
   a) instrument table name
   b) entity name (instrTypeEncoded)
   c) scope (used to distinguish variables at core, crms and app scope)

   e.g. call util_AddTableToMetaData('sf_36', 'sf36', 'app-smd')

6) In the previous steps, the viewproperty table/property column and the hibernateproperty table/property and hibernateProperty columns are populated with the variable column name . These column values represents the Java class property name for the instrument, so if you are using a naming convention where table column names use '_' as separators, and would like the corresponding Java property names to instead capitalize each section of the variable name, then you can execute the util_FixMetadataPropertyNames utility stored procedure.

   e.g. to convert one_big_var to oneBigVar for instrument 'sf36', execute it twice, one for each underscore:
   call util_FixMetadataPropertyNames('sf36')
   call util_FixMetadataPropertyNames('sf36')

7) At this point, the viewproperty table needs to be completed. A row has been inserted for each instrument property, but a number of columns were not populated. In particular, the following columns may need to be populated:

style	style of the input control used. see createField.tag documentation and source for more info. most common styles are: 'scale' for a variable domain with <= 5 entries (radio buttons) 'range' for variable domain with > 5 entries (dropdown box) but is often used instead of 'scale' for <= 5 entries when a dropdown box is desired 'suggest' is the same as a range but also allows the user to input their own text if the value they want is not in the dropdown list 'string' is for single line of free form text (used in conjunction with the viewproperty size and maxLength columns, see textBox.tag) 'text' is for multiple lines of text, i.e. notes fields (used in conjunction with the viewproperty maxLength column, see textarea.tag) 'date' is for a date control 'toggle' is for a checkbox for an underlying Boolean or Integer/Short property
context	typically 'r' for an instrument variable, i.e. a variable that is a standard part of an assessment and is required (if it is a free form string field that is optional and can not be compared using instrument double entry then use 'i') if it is a field that is calculated by the model object (i.e. not input by the user), then use 'r' in conjunction with 'disabled' in the attributes column and 'ref.totalErrorCode' in the list column if it is a non-editable field, e.g. a calculated field, an alternative to the above approach is to just set context to 'c' otherwise if it is an editable field and 'r' does not apply, use 'i' see createField.tag documentation and source for more info.
list	if the variable input control requires a list, this is the list identifier, where the list is defined within a WEB-INF/context configuration xml file the streamlined creation scripts auto-populate this with the first 50 chars of the data_values column in the datadictionary xlsx, as a reminder of what list to use. you need to replace this text with a list. note: list items can be in the database (list/listvalues tables) or configured directly within the xml configuration file. See lava-crms/WEB-INF/context/crms/lists for examples of the former. See lava-crms-

	nacc/WEB-INF/context/crms/nacc-lists.xml for examples of the latter (e.g. list.uds.b3.bradykin). lists are often built using other lists as building blocks, i.e. specifying a parent list. The defaultCodes list configuration property is used to add the instrument standard error codes and/opr skip codes into a list and references the pre-existing codes lists in lava-crms/WEB-INF/context/crms/lists/lists-codes.xml there are some general purpose lists already configured, e.g. use ‘generic.yesNoZero’ to present Yes/No options that store 0/1 in the database. see lava-core/WEB-INF/context/core/lists/lists-common.xml for this and other common lists numeric and decimal range lists are easy to configure, e.g. use parent=”list.numericRange” and specify the defaultMinValue and defaultMaxValue properties. See lava-crms-nacc/WEB-INF/context/crms/nacc-lists.xml for examples. the lists for instruments are typically configured in the same Spring configuration file used to configure other aspects of the instrument (see Spring configuration below). note: the “list.” prefix in the list definition is omitted in the viewproperty list column
label	label for variable, this is either the same as the prop_description column in the datadictionary xlsx, or a shortened version of it (the streamlined creation scripts auto-populate this with the prop_description)
label2	a second label, if needed, e.g. the underlying database column name to precede the variable input control, where label follows the input control
quickHelp	description for variable, could be same as label or expanded. This is the same as the prop_description column in the datadictionary xlsx (the streamlined creation scripts auto-populate this with the prop_description)
attributes	any HTML attributes for the variable input control, e.g. for style=”text” could define “rows” and “cols”
maxLength	only applies to those styles where the user can input text such as ‘string’ and ‘text’. see createField.tag documentation and source for more info. this the same as the db_datalength column in the datadictionary xlsx (the streamlined creation scripts auto-populate this with the db_datalength)
size	can set the size of the input control, in characters. see createField.tag documentation and source for more info.
required	this is ignored for intrument properties, so can be null or ‘Yes’. for instrument properties, context=’r’ designates a required variable

8) “Code” generation

Now that hibernateproperty and viewproperty are populated, the util_GenerateCode stored proc can be executed to generate code to assist in creating the various building blocks needed to implement the instrument. In order for the output to format properly, in MySQL, the proc should be run using the mysql command line tool with certain switches:

For example, create a sf36.sql script file for mysql to run, containing the following 2 lines:

```
use smd;
call util_GenerateCode('sf36','app-smd');
```

Execute the script using the mysql command line tool:

```
mysql -s -r -u DB_USER -p < sf36.sql > gen_sf36.txt
```

The output file (gen_sf36.txt in this example, gen_code.sql in the streamlined scripts instructions) contains “code” to be used in the creation of the instrument and will be used in Step 11) below.

9) LavaQuery View. The util_CreateLQView stored procedure generates the database view required for LavaQuery to download the instrument data. You will need to enter the following parameters:
   a) table name (database table name)
   b) instrument name (instrTypeEncoded)

   Execute the resulting code into your database. For example, create a gen_lavaquery.sql script file for mysql to run, containing the following 2 lines:

   ```
   use smd;
   call util_CreateLQView('sf36','SF-36');
   ```

   Execute the script using the mysql command line tool:

   ```
   mysql -s -r -u DB_USER -p < gen_lavaquery.sql > lavaquery_code.sql
   ```

10) LavaQuery Proc. The util_CreateLQProc stored procedure generates the stored procedure required for LavaQuery to interact with the database depending on what parameters the user enters in LavaQuery. You will need to enter the following parameters:
   a) scope
   b) section
   c) target
   d) instrument name (instrTypeEncoded)
   e) table name (database table name)

   Execute the resulting code into your database. For example, create a gen_lavaquery.sql script file for mysql to run, containing the following 2 lines:

   ```
   use smd;
   call util_CreateLQView('crms','crms_smd','sf36','SF-36');
   ```

   Execute the script using the mysql command line tool:

   ```
   mysql -s -r -u DB_USER -p < gen_lavaquery.sql > lavaquery_code.sql
   ```

Streamlined Creation Scripts

Steps 2) thru 10) above have been condensed into a set of scripts that can be combined with a set of variable definitions for the new instrument that can further automate the instrument creation process. Given an understanding of the underlying steps involved in creating a new instrument, one can instead use the following scripts instead of 2) thru 10) above.

These scripts require completing Step 1) above to populate the datadictionary xlsx file for your new instrument.

You will find these scripts in
/lava-crms/development/instruments
You will also find an example of the scripts setup for creating a specific instrument called 'SCQ', along with the output files generated by the scripts and a populated datadictionary_scq.xlsx.
/lava-crms/development/instruments/exampleSCQ

0_shell_script.sql (shows the syntax to use to run scripts 2_ thru 6_ below)
Replace DB_USER with your database user.

1_set_vars.sql (sets the variables for your specific instrument)
All of the following scripts source this file to determine the name of the database, the name of the instrument (@instrname, which is the display name), name of the entity (@entity, which is instrTypeEncoded for instruments) and associated values specific to the instrument being created.

2_gen_table.sql
This script both inserts rows into the database and generates an output file.

Prior to execution you will need to edit this file, copying & pasting in column C of the datadictionary xlsx to the place indicated in the file. These are the insert statements for the datadictionary table.

Upon execution the script inserts all of the properties into the datadictionary table and then uses this datadictionary data to generate a SQL script containing the CREATE TABLE statement for the instrument, an ALTER TABLE statement for creating the instrument foreign key, and an INSERT statement to insert the instrument name into the "instrument" table.

Now execute the newly generated gen_table.sql to create the table for the new instrument in the database, as well as populate the "instrument" table (this is the table with one row for each type of instrument in the database, which is used to populate a dropdown of instrument types when a user adds an instrument).

3_insert_metadata.sql
This script inserts rows into the database but does not output anything.

Property metadata for each property of the new instrument is inserted into the viewproperty and hibernateproperty tables. This metadata is used by the next script, 4_gen_code.sql, to generate view-related and persistence related "code".

Following execution of 3_insert_metadata.sql and prior to executing 4_gen_code.sql, the viewproperty table needs to be completed for the instrument. The following will get all of the pertinent rows, given instrTypeEncoded = ‘scq’ and scope = ‘app-xyz’
select * from viewproperty where scope=’app-xyz’ and entity = ‘scq’;

Complete the population of viewproperty for these properties as described in Step 7) above.

4_gen_code.sql
This script does not insert anything into the database but generates an output file.

The output file, gen_code.sql, is used when continuing with Step 11) below, following use of the streamlined creation scripts. Step 11) details what code is generated and how it is used.

5_gen_metadata.sql
This script does not insert anything into the database but generates an output file.

The gen_metadata.sql output script is not used directly in the creation of the instrument. It is used when releasing the instrument to another (production) database, and to manage multi-programmer development. The script is described in the “Metadata / List Data” subsection of this document.

6_gen_lavaquery.sql
This script does not insert anything into the database but generates an output file.

The output file gen_lavaquery.sql contains creation statements for the SQL view and SQL stored procedure which support retrieving data from the instrument using the LAVA Query tool (these are specific to MySQL). The output file also contains an INSERT statement to insert a LAVA Query object for the instrument into the query_objects table, which will make it available to users of the tool.

Now execute the gen_lavaquery.sql output file. Following the completion of all the instrument creation steps your instrument will be LAVA Query ready.

Streamlined Creation Script Notes
It is recommended that you create a folder for each instrument and copy the scripts into the folder. Then set the variables in 1_set_vars.sql and execute scripts as outlined in 0_shell_scripts.sql. The scripts are command-line scripts to be executed using the mysql shell.
END: Streamlined Creation Scripts Usage

11) Use the code generation output file from above (either the output file from Step 8), or if using the Streamlined scripts, the gen_code.sql generated by 4_gen_code.sql) to copy & paste to create the following artifacts for your new instrument:
    - Hibernate mapping file

- Java model class with properties and getRequiredResultField method. After the properties are copied & pasted to a Java class, the Eclipse Source generation can be used to "Generate Getters and Setters" for the properties
- jsp file

note: the buffer.append(UdsUploadUtils..) generated code is only applicable for NACC UDS instruments.

You can reference an existing instrument to make sure you create these artifacts properly following instructions below. You can also copy & paste the corresponding files from an existing instrument as a starting point and then replace the code with the generated code from gen_code.sql.

Using the MdsStatus instrument from the LAVA Demo app as an example, the corresponding files would be:
/lava-crms-nacc/WEB-INF/hibernate/crms/assessment/MdsStatus.hbm.xml
/lava-crms-nacc/src/edu/ucsf/lava/crms/assessment/model/MdsStatus.java
/lava-crms-nacc/WEB-INF/jsp/crms/assessment/instrument/mdsstatus.jsp

note: unless you are creating an instrument that needs to be uploaded to the NACC you do not need to implement UdsUploadable as MdsStatus does, nor do you need the getUdsUploadCsvRecord(s) methods.

Following are the details of creating each artifact.

<u>Java class file</u>
Instrument classes go into the following package in your app's src directory:
edu.ucsf.lava.crms.assessment.model

Use the shell of an existing instrument Java class file and do the following:

- name the class. An instrument Java class should be named after the "instrType" with spaces and non-alphanumeric characters removed and the initial character of each part of the name capitalized, e.g. "UdsFaq"
- copy & paste the code-generated properties declarations into the class and use Eclipse Source "Generate Getters and Setters" to create getters/setters
- copy the code-generated property name strings to the getRequiredResultFields method.

note: getRequiredResultFields defines instrument variables as required. A user will not be allowed to complete data entry without entering data for all required fields. Standard Error codes can be used to for a variable that does not have an actual value to enter

properties with style 'date' should not be required unless guaranteed to have a value because the standard error and skip codes can not be assigned to this data type, so do not include these in the getRequiredResultFields method (these properties still have context 'r' because they are instrument result fields and should be compared in instrument double enter)

properties that are calculated fields should also be removed from getRequiredResultFields because they will not have a value until the instrument is saved note: if the instrument has calculated properties, override the following method in the instrument's Java class file to do any calculations:

```
public void calculate() throws Exception;
```

<u>Hibernate mapping file</u>

Instrument mapping files are located in your app in the directory:
WEB-INF/hibernate/local/APP_NAME/crms/assessment

Either copy & paste the entire mapping output (from the very top of the code generated file to the "</hibernate-mapping>" or use the shell of an existing instrument Hibernate mapping file (e.g. from lava-crms-nacc) and ensure the following:

- name the mapping file the same as the instrument Java class name
- in the <class> element, set name to the instrument fully qualified Java class name
- set the <join> element table to the instrument table name
  note: in the <class> element, table="instrumenttracking" for all instruments
- copy the code generated properties within the <join> element

<u>jsp file</u>

The instrument jsp files are located in your app in the directory:
jsp/crms/assessment/instrument

Start with the shell of another instrument jsp, and do the following:

- name the jsp file after instrTypeEncoded with a .jsp extension
- set the instrTypeEncoded JSTL variable at the top of the file
- set focusField accordingly
- copy code-generated createField statements

  a number of variations of createField and listField statements are generated for the instrument jsp file. The default group to use for instruments is:
  <tags:createField property="_prop_" component="${component}" entity="${instrTypeEncoded}"/>

  The <tags:tableForm> generated output is an alternative layout that presents the instrument form in a tabular format with labels in one column and input in another. This may be used so that the LAVA instrument form more closely approximate a paper form. The label column can include instruction text and/or scale definitions. The code generation presents two variations on this format, first with labels to the left, and second with labels to the right.

- add/delete sections and quicklinks
  note: section sectionNameKey and quicklink keys go into the app's messages.properties file (quicklink keys are optional but can be created if the sectionNameKey value is too long for a quicklink. Just add an additional message key replacing .section with .quicklink)

- add skip logic to the jsp via formGuide tags (see instrument jsps in lava-crms-nacc for formGuide examples and reference the uitags.tld file for documentation of the formGuide tag)

12) Spring configuration.
Create a Spring context configuration file for all of the instruments in your app (or you could create several config files if you instruments should be grouped in a certain way). This file lives in the following directory in your app:
WEB-INF/context/local/APP_NAME/APP_NAME-instruments.xml

The actually filename is up to you, but it must be included by the APP_NAME-context.xml file in the same directory.

For example instrument configurations, see the lava-crms-nacc
WEB-INF/context/crms/nacc-instruments.xml

This file will configure the following for the instrument:

- an "action" bean (actions are covered in the "LAVA Web Application Development Framework Introduction") using the naming:
lava.crms.assessment.instrument.[instrTypeEncoded]

- a FormAction bean (FormAction is a Spring Web Flow construct) named by concatenating instrTypeEncoded and "FormAction"

  note: unless the instrument has complex requirements such that it needs its own handler for events, the FormAction bean will just define:
parent="instrumentFormAction"

- an instrumentConfig bean defining the Java model class defined for the instrument and which flows the instrument supports, among other things.

  See lava-crms/WEB-INF/context/crms/crms-instrument.xml for documentation on the properties of an instrumentConfig bean. Note that the definition of a "instrumentConfigBeanPostProcessor" bean utilizes the Spring frameworks bean postprocessing functionality to allow configuration of instrumentConfig beans where the bean definition is independent and not part of any larger bean structure.

- a .codes list for the instrument, which determines which skip and/or standard error codes are presented as potential values for an instrument variable when such a value is needed
See lava-crms-nacc/WEB-INF/context/crms/nacc-lists.xml for .codes examples

- list configurations for any variables of the instrument that have an associated list (the list is associated with an instrument property in the viewproperty table)

  list configurations may also be in a separate Spring configuration file as is done in lava-crms-nacc (nacc-lists.xml). just make sure the config file is included by the context.xml file

13) Add a metadata record to the "instrument" table. The instrument must be in this table to appear in the "Add Instrument" dropdown list (if using the streamlined scripts this INSERT

statement was generated in the gen_table.sql file, so if that was executed as directed you can skip this step)

e.g.

```
insert into instrument(InstrName, TableName, FormName)
values('SF-36', 'sf_36', 'LavaWebOnly');
```

note: for FormName, always use “LavaWebOnly”

14) Add a LavaQuery record to the “query_objects” table. This must be added for each instrument to be available in LavaQuery (if using the streamlined scripts this INSERT statement was generated in the gen_lavaquery.sql file, so if that was executed as instructed you can skip this step)

e.g.

```
INSERT INTO query_objects(instance,scope,module,section,target,short_desc,standard,primary_link,secondary_link)
VALUES('lava','crms','query','app_smd','smddiagnosis', 'SMD Diagnosis',1,1,1);
```

Here are explanations for some of the columns:

a) standard (whether the instrument should be available for standard download on the Database worksheet of LavaQuery.
b) primary_link (whether the instrument should be available as a primary link on the InstrumentGrouping and VisitGrouping worksheets of LavaQuery.
c) secondary_link (whether the instrument should be available as a secondary link on the InstrumentGrouping and VisitGrouping worksheets of LavaQuery.

15) Compile, deploy/load and debug your new instrument.

LAVA Error Codes

LAVA instrument properties are always required. In the case where a property does not have a "valid" value LAVA provides two sets of codes which should be used to indicate why there is not a value. LAVA will not allow completion of instrument data entry if any properties are left blank.

Note: it is possible to have instrument properties that are not required, such as optional notes

The "Standard" error codes apply to all instruments, while the "Skip" error codes may only be applicable for some instruments. Lists for instrument properties are configured accordingly.

Standard Error Codes

Number	Label	Description
-9	Missing	This code is used to indicate the data item is missing. This code has been applied retrospectively to old records where the reason for missing data could not be determined. It can also be used when entering old versions of an instrument that are missing new data fields used on current instrument data entry screens.
-8	Unused Variable	This code is used to indicate that a field is not used by the instrument. In particular this code is automatically entered into instrument fields that have been discontinued
-7	Incomplete Data Entry	This code is used to indicate that the data for the field has not yet been entered.
-6	Logical Variable Skip	This code is used to indicate that a data item is not required because of the "logical flow" of the instrument.

Skip Error Codes

Number	Label	Description
-4	Refused	This code is used to indicate that the data item was not collected because the patient refused
-3	Alternate Test Given	This code is used to indicate that the data items were not collected because and alternate test was used. Typically only used for those instruments that consist of multiple sub-tests.
-2	Situational Factor	This code is used to indicate that the data item was not collected because of a factor such as running out of time, or having the testing interrupted.
-1	Patient Factor	This code is used to indicate that the data item was not collected because of a factor related to the patient's (or informant's) capacity (e.g. MMSE to low).

Instrument Skip Logic

Skip logic was covered in the “Skip Logic” section of the “Extending and Customizing” section. This section augments that with information specific to instrument skip logic.

Following is a usage example from lava-crms-nacc WEB-INF/jsp/crms/assessment/instrument/udshachinski2.jsp

```
<!—do not need skip logic in ‘vw’ mode, i.e. readonly view →
<c:if test="${componentMode != 'vw'}">
	<!—this loop is specific to instrument skip logic and supports instrument double entry. on the double entry comparison page
	two sets of the instrument properties are displayed, so need skip logic working on each set of properties →
	<c:forEach begin="0" end="1" var="current">
		<c:choose>
			<c:when test="${componentView == 'doubleEnter' || (componentView == 'compare' && current == 1)}">
				<c:set var="component" value="compareInstrument"/>
			</c:when>
			<c:otherwise>
				<c:set var="component" value="instrument"/>
			</c:otherwise>
		</c:choose>
		<c:if test="${current == 0 || (current == 1 && componentView == 'compare')}">
			<ui:formGuide>
				<!—if cvsImag is Yes=1 then unskip a number of dependent properties. Note that regexp “^1” is used to make
				sure that a -1 will not match. Also note this differs from regexp “[^1]” which means matching anything but 1.

				tags also do the inverse action (unless ignoreUndo and/or ignoreUndoOnLoad are specified for the formGuide
				tag) so when cvsImag is not Yes=1 then the dependent properties are unskipped.

				ui:skip and ui:unskip actions are specific to instrument usage since ui:skip not only disables a field but sets its
				value to the “Logical Variable Skip” standard error code -6, while ui:unskip enables a field and if that field had
				the value -6 its value is reset to null.

				comboRadioSelect should be used for any properties which could be using what is known as a
				comboRadioSelect control, as determined by createField. (specifically when in direct collection mode=’dc’
				and style=’scale’) →

				<ui:observe elementIds="cvdImag" component="${component}" forValue="^1"
						comboRadioSelect="${componentMode == 'dc' ? 'true' : 'false'}"/>
				<ui:unskip elementIds="cvdImagSingle,cvdImagMultiple,cvdImagExtensive,cvdImagOther"
						component="${component}" comboRadioSelect="${componentMode == 'dc' ? 'true' : 'false'}"/>
			</ui:formGuide>

			<!—the final formGuide tag on a page should include the following simulateEvents attribute so that all formGuide
			tags on the page are evaluated on page load. →
			<ui:formGuide simulateEvents="${(current == 0 && componentView != 'compare') || (current == 1) ? 'true' : ''}">
				<!—the ui:depends tag ensures that this tag fires when the cvsImag property value changes →
				<ui:depends elementIds="cvdImag" component="${component}"
						comboRadioSelect="${componentMode == 'dc' ? 'true' : 'false'}"/>
				<!—when the cvsImagOther property is Yes=1 then unskip the cvsImagOtherX property (and vice versa) →
				<ui:observe elementIds="cvdImagOther" component="${component}" forValue="^1"
						comboRadioSelect="${componentMode == 'dc' ? 'true' : 'false'}"/>
				<ui:unskip elementIds="cvdImagOtherX" component="${component}"/>
			</ui:formGuide>
		</c:if>
	</c:forEach>
</c:if>
```

Creating Visit Instrument Prototypes

To automatically create and associate a list of instruments with a visit type the localInstrumentPrototypes configuration is used. This configuration maps a key to a list of instrument classes, where the key is a combination of a Project and a Visit Type. When the user adds a Visit entity where the projName and visitType properties match a prototype configuration, the listed instruments are automatically created and associated with the visit that is added.

Using the lava-app-demo as an example, localInstrumentPrototypes is defined in:
WEB-INF/context/local/demo/demo-scheduling.xml

The following configuration results in creation of a full set of the UDS instruments when a user adds a Visit with projName “UDS” and visitType “Initial Assessment”

```
<bean id="localInstrumentPrototypes" class="java.util.LinkedHashMap"><constructor-arg><map>
	<entry key="UDS~Initial Assessment"><list>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsAppraisal" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Appraisal"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsCdr" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS CDR"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsDiagnosis" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Diagnosis"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsFamilyHistory2" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Family History"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsFaq" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS FAQ"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsGds" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS GDS"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsHachinski" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Hachinski"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsHealthHistory" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Health History"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsInformantDemo" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Informant Demo"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsLabsImaging" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Labs Imaging"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsMedications2" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Medications"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsNeuroPsych" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS NeuroPsych"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsNpi" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS NPI"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsPhysical" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Physical"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsSubjectDemo" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Subject Demo"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsSymptomsOnset" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Symptoms Onset"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsUpdrs" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS UPDRS"/>
		</bean>
		<bean class="edu.ucsf.lava.crms.assessment.model.UdsFormChecklist" parent="initialUDSInstrumentPrototype">
			<property name="instrType" value="UDS Form Checklist"/>
		</bean>
	</list></entry>
```

Calculations

If an instrument requires calculations such as totals or some other statistics, LAVA has a hook for where such calculations should go. Every instrument model class inherits (directly or indirectly) from the Instrument class. Instrument has an updatedCalculatedFields method which is invoked right before an instrument is saved. Instruments that have calculations should override this method in their model classes, perform all calculations and assign the results to calculated properties. It is important that the method invoke super.updateCalculatedFields because the Instrument class implementation of the method contains vital functionality.

There is also a controller in lava-crms called the CalculateController that is a developer utility to facilitate recalculating a specific instrument or all existing instruments of a specific type due to a change in the calculations. Consult the comments in the source code for how to go about using this.
edu.ucsf.lava.crms.assessment.controller.CalculateController

Data Double Entry

LAVA supports double entry of instrument data for data quality purposes. With the double entry feature, after the data has been entered for all instrument fields the user can choose “Double Entry” to enter the data again. Upon saving following the second data entry the user will be returned to the data entry form if there are any differences between the first and the second data entry. The user must reconcile the differences and will not be permitted to save until all entries match.

Double Entry is only available for instruments that are configured such that the “verify” property of the instrumentConfig definition for the instrument is true (which is the default).

Only the instrument fields that are result fields are involved in the comparison, where result fields are designated with the value ‘r’ in the ViewProperty metadata “style” column.

Double Entry can be configured such that the user is taken to the data double entry form after completion of the data entry. The configuration is defined as a percentage of the time that the user must double enter the instrument data. However, at this time the only percentage supported is 100 which essentially configures an instrument for mandatory double entry. Note that even with this setting a user has the option to “Defer” double entry, but it is effective in presenting the user with the double entry form every time they complete data entry for an instrument of a given type. In order to configure mandatory double entry for an instrument, use the localProjectInstrumentVerifyRates bean.

e.g. to require mandatory double entry for the “ABC” assessment (instrument) in the “Normals” study (project), configure as follows, using “abc” for the instrTypeEncoded value of the “ABC” instrument:

```
<bean id="localProjectInstrumentVerifyRates" class="java.util.LinkedHashMap">
        <constructor-arg><map>
            <entry key="Normals~abc" value="100"/>
        </map></constructor-arg>
</bean>
```

To require mandatory double entry for all instruments across all projects, use the “ANY” keyword to match all projects and all instruments, as follows:

```
<bean id="localProjectInstrumentVerifyRates" class="java.util.LinkedHashMap">
        <constructor-arg><map>
            <entry key="ANY~ANY" value="100"/>
        </map></constructor-arg>
</bean>
```

Data Review

Some instruments may not work well with double entry, but should still be verified with a visual review. To achieve this take the following two steps:

1) The instrumentConfig bean “verify” property should be set false.
2) The instrumentConfig bean “supportedFlows” property should include “enterReview”

Now, after the user completes data entry of the instrument, they must click “Review” and are taken to a readonly view of the instrument. Upon visual inspection that the instrument data is correct, the user can then save the instrument.

Customizing Instrument Status

Every instrument in LAVA has an instrument status, which is metadata pertaining to when the data for the instrument was collected, when it was entered and when it was verified. The status form is part of the standard instrument flow, i.e. the instrument status form is always displayed after an instrument is data entered, with some of the status values imputed where it is logical to do so.

LAVA has a mechanism for overriding the default instrument status form for the application as a whole, e.g. if certain of the default status properties are never used such that they should not be displayed.

In the instrument content decorator file there is the following HTML snippet which determines the file location of the instrument status content file:

```
<c:set var="statusContentURL">
<spring:message code="${webappInstance}.instrumentStatusContentURL"
        text="crms/assessment/instrument/statusContent.jsp"/></c:set>
```

The default location for the instrument status content is specified in the text attribute: WEB-INF/jsp/crms/assessment/instrument/statusContent.jsp

To override this with a customized instrument status content file:

1) In the custom i18n properties file for the application, define the file location of the custom content file. e.g. for application instance "xyz" define the following in the app-xyz-custom.properties file:
   xyz.instrumentStatusURL=local/xyz/crms/assessment/instrument/statusContent.jsp

2) Create the above custom instrument statusContent.jsp file which will replace the default lava-crms statusContent.jsp

Customizing NACC UDS Patient ID

The value used for the NACC UDS patient ID in submissions of the UDS data set to the NACC can be easily customized in LAVA. The default is to supply the internally generated LAVA PIDN as the ID. In LAVA this ID is stored in the "ptid" property of the UdsInstrument class, which all of the UDS instruments subclass.

The mechanism used to customize the ID is an ID strategy pattern built into core LAVA infrastructure. Customizing the ID essentially means providing a new strategy for obtaining the ID.

Following is an example of application "xyz" customizing the ID to use the "naccId" property on the Patient entity (where "naccId" is an extended property of XyzPatient). There are two steps involved.

1) Override the UdsInstrumentPtidStrategy class and override the getEntityPropId method.
   edu.ucsf.lava.local.xyz.crms.id.XyzUdsInstrumentPtidStrategy

```
package edu.ucsf.lava.local.xyz.crms.id;
…
public class XyzUdsInstrumentPtidStrategy extends UdsInstrumentPtidStrategy {
        public String getEntityPropId(EntityBase entity, String idPropName) {
                UdsInstrument instr = (UdsInstrument) entity;
                return ((XyzPatient) instr.getPatient()).getNaccId();
        }
}
```

2) Redefining the "udsInstrumentPtidStrategy" bean to use the new strategy class.
   WEB-INF/context/local/xyz/xyz-local.xml

```
<bean id="udsInstrumentPtidStrategy" class="edu.ucsf.lava.local.xyz.crms.id.XyzUdsInstrumentPtidStrategy"/>
```

NOTE: because of the order in which context files are imported in webapp deployment, the app level beans will be last and will override any beans of the same name in lava-core, lava-crms and lava-crms-nacc. So the “udsInstrumentPtIdStrategy” bean definition replaces its bean definition in lava-crms-nacc.