Ada Web Application Programmer's Guide

STEPHANE CARREZ

2020-04-05

Contents

1 Introduction

Ada Web Application is a framework to build a Web Application in Ada 2012. The framework provides several ready to use and extendable modules that are common to many web applications. This includes the login, authentication, users, permissions, managing comments, tags, votes, documents, images. It provides a complete blog, question and answers and a wiki module.

AWA simplifies the Web Application development by taking care of user management with Google+, Facebook authentication and by providing the foundations on top of which you can construct your own application. AWA provides a powerful permission management that gives flexibility to applications to grant access and protect your user's resources.

A typical architecture of an AWA application is represented by the picture below:

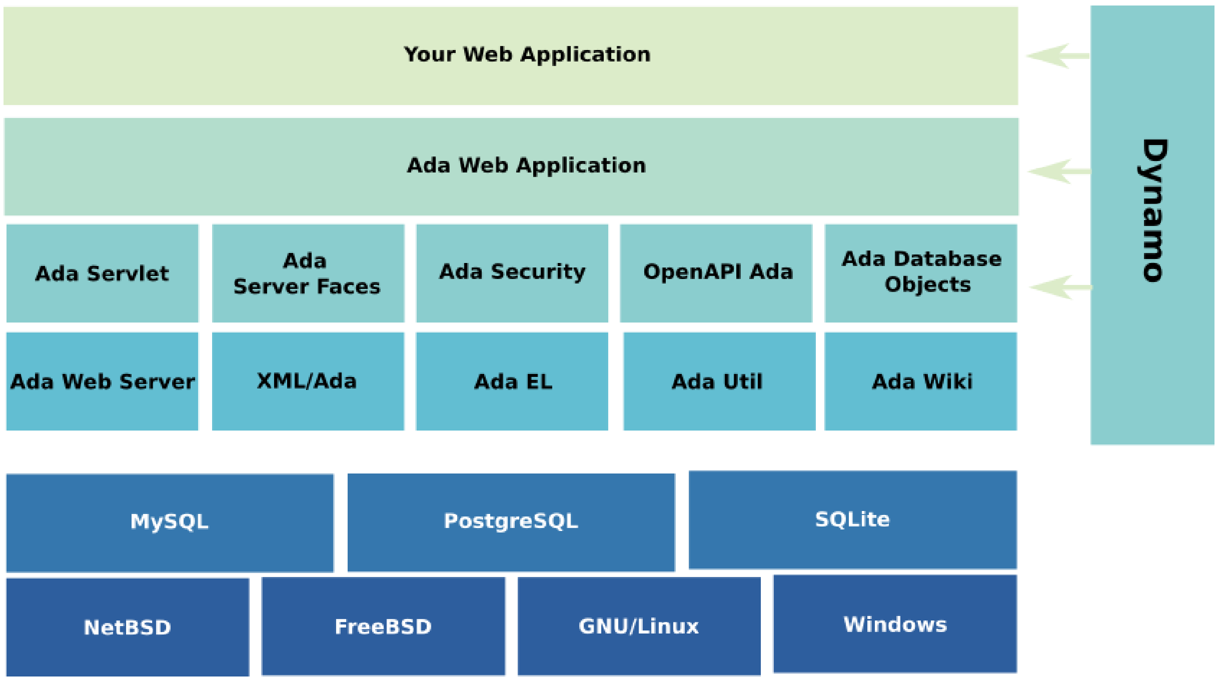


Figure 1: *Ada Web Application Architecture*

Because your application sits on top of AWA framework, it benefits of all the functionalities that AWA uses for its implementation:

- The Web server is built on top of the Ada Web Server library,
- The presentation layer is using Ada Server Faces which allows to use the same design pattern as the Java Server Faces,
- The database access is provided by Ada Database Objects

Apart from this architecture, the Dynamo tool is used to generate code automatically and help starting the project quickly.

AWA is composed of several configuration components also called modules or plugins. Components are classified in three categories:

- System components,
- General purpose components,
- Functional components.

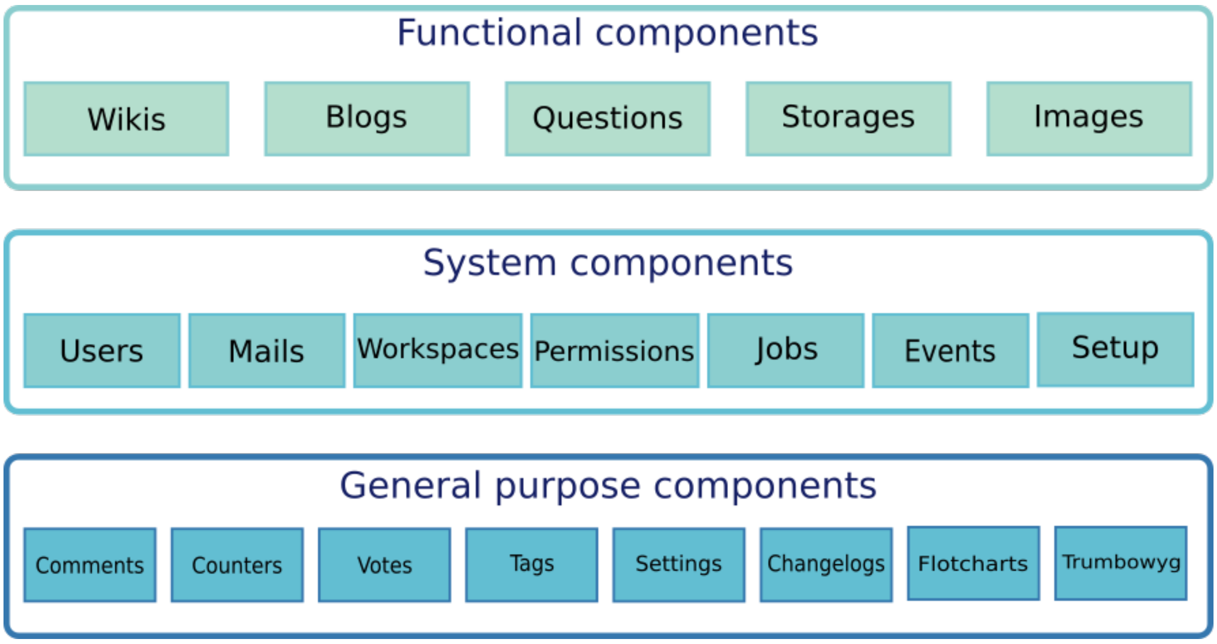


Figure 2: *AWA Features*

1.1 System Components

The **System Components** represent the core components onto which all other components are based. These component don't provide any real functionality for a final user but they are necessary for the Web application to operate. These components include:

- The Users Module manages the creation, update, removal and authentication of users,
- The Mail Module allows an application to format and send a mail,
- The Jobs Module provides a batch job framework for modules to execute long running actions,
- The Events Module implements an eventing system to share events with other modules,
- The Workspace Module defines a workspace area for other plugins to connect and plug into.

1.2 General Purpose Components

The **General Purpose Components** are components which provide generic functionalities that can be plugged and used by functional components.

- The Tags Module allows to associate general purpose tags to any database entity,
- The Votes Module allows users to vote for objects defined in the application,
- The Comments Module is a general purpose module that allows to associate user comments to any database entity,
- The Counters Module defines a general purpose counter service allowing to associate counters to database entities,
- The Changelogs Module associates logs produced by users to any database entity.

1.3 Functional Components

The **Functional Components** implement a final functionality for a user. They are using the system components such as User Module for the user management but also general purpose components such as Tags Module or Counters Module.

- The Questions Module is a simple question and answer system,
- The Blogs Module is a small blog application which allows users to publish articles,
- The Wikis Module provides a complete Wiki system allowing users to create their own Wiki environment.

To help in the installation process of final applications, the Setup Application is a special component that you can decide to customize to provide an installation and configuration process to your own application.

2 Installation

This chapter explains how to build and install the Ada Web Application framework.

2.1 Before Building

Before building the framework, you will need:

- The GNAT Ada compiler,
- Either the MySQL, PostgreSQL or SQLite development headers installed,
- XML/Ada,
- Ada Web Server.

First get, build and install the above tools and libraries. For the best experience, it is necessary to have the SSL support in Ada Web Server. Indeed, the OpenID Authentication 2.0 can only be used through HTTPS.

The build process may also need the following commands:

- make (GNU make),
- gprbuild,
- gprinstall,
- unzip,
- sqlite3,
- mysql,
- psql,
- xsltproc,
- liblzma libraries (used by Ada LZMA),
- CURL libraries (used by CURL support in Ada Utility Library)

The Ada Web Application library also uses the following projects:

- Ada LZMA,
- Ada Utility Library,
- Ada Expression Language Library,
- Ada Security Library,
- Ada Servlet Library,
- Ada Server Faces Library,
- Ada Wiki Library,
- Ada Database Objects Library,
- Ada Keystore Library,

- OpenAPI Ada Library,
- Dynamo

They are integrated as Git submodules.

2.2 Getting the sources

The AWA framework uses git submodules to integrate several other projects. To get all the sources, use the following commands:

```
git clone --recursive git@github.com:stcarrez/ada-awa.git
cd ada-awa
```

2.3 Development Host Installation

The PostgreSQL, MySQL and SQLite development headers and runtime are necessary for building the Ada Database Objects driver. The configure script will use them to enable the ADO drivers. The configure script will fail if it does not find any database driver.

2.3.1 Ubuntu

First to get the LZMA and CURL support, it is necessary to install the following packages before configuring AWA:

```
sudo apt-get install liblzma-dev libcurl4-openssl-dev
```

MySQL Development installation

```
sudo apt-get install libmysqlclient-dev
```

MariaDB Development installation

```
sudo apt-get install mariadb-client libmariadb-client-lgpl-dev
```

SQLite Development installation

```
sudo apt-get install libsqlite3-dev
```

PostgreSQL Development installation

```
sudo apt-get install postgresql-client libpq-dev
```

2.3.2 FreeBSD 12

First to get the LZMA, XML/Ada and CURL support, it is necessary to install the following packages before configuring AWA:

```
pkg install lzma-18.05 curl-7.66.0 xmlada-17.0.0_1 aws-17.1_2
```

MariaDB Development installation:

```
pkg install mariadb104-client-10.4.7 mariadb104-server-10.4.7
```

SQLite Development installation:

```
pkg install sqlite3-3.29.0
```

PostgreSQL Development installation:

```
pkg install postgresql12-client-12.r1 postgresql12-server-12.r1
```

Once these packages are installed, you may have to setup the following environment variables:

```
export PATH=/usr/local/gcc6-aux/bin:$PATH
export ADA_PROJECT_PATH=/usr/local/lib/gnat
```

2.3.3 Windows

It is recommended to use msys2 available at https://www.msys2.org/ and use the `pacman` command to install the required packages.

```
pacman -S git
pacman -S make
pacman -S unzip
pacman -S base-devel --needed
pacman -S mingw-w64-x86_64-sqlite3
```

For Windows, the installation is a little bit more complex and manual. You may either download the files from MySQL and SQLite download sites or you may use the files provided by Ada Database Objects and Ada LZMA in the `win32` directory.

For Windows 32-bit, extract the files:

```
cd ada-ado/win32 && unzip sqlite-dll-win32-x86-3290000.zip
cd ada-lzma/win32 && unzip liblzma-win32-x86-5.2.4.zip
```

For Windows 64-bit, extract the files:

```
cd ada-ado/win32 && unzip sqlite-dll-win64-x64-3290000.zip
cd ada-lzma/win32 && unzip liblzma-win64-x64-5.2.4.zip
```

If your GNAT 2019 compiler is installed in `C:/GNAT/2019`, you may install the liblzma, MySQL and SQLite libraries by using msys cp with:

```
cp ada-lzma/win32/*.dll C:/GNAT/2019/bin
cp ada-lzma/win32/*.dll C:/GNAT/2019/lib
cp ada-lzma/win32/*.a C:/GNAT/2019/lib
cp ada-ado/win32/*.dll C:/GNAT/2019/bin
cp ada-ado/win32/*.dll C:/GNAT/2019/lib
cp ada-ado/win32/*.lib C:/GNAT/2019/lib
cp ada-ado/win32/*.a C:/GNAT/2019/lib
```

2.4 Ada Web Server

The Ada Web Server should be compiled with the SSL support if you want to use the OAuth 2.0 protocol and integrate with Google or Facebook authentication systems. The AWS version shipped with GNAT 2019 and GNAT 2020 will not work because it does not support SSL.

You may build AWS by using:

```
   git clone --recursive -b 20.2 https://github.com/AdaCore/aws
   cd aws
   make SOCKET=openssl setup build install
```

2.5 Configuration

The library uses the `configure` script to detect the build environment, check for Ada Utility Library library. The `configure` script provides several standard options and you may use:

- `--prefix=DIR` to control the installation directory,
- `--enable-shared` to enable the build of shared libraries,
- `--disable-static` to disable the build of static libraries,
- `--enable-distrib` to build for a distribution and strip symbols,
- `--disable-distrib` to build with debugging support,
- `--enable-coverage` to build with code coverage support (`-fprofile-arcs -ftest-coverage`),

- `--with-aws=PATH` to control the installation path of Ada Web Server,
- `--with-xmlada=PATH` to control the installation path of XML/Ada,
- `--help` to get a detailed list of supported options.

In most cases you will configure with the following command:

```
./configure
```

By default, the framework will be installed in `/usr/local` directory. If you want to install the framework in a specific directory, use the `--prefix` option as follows:

```
./configure --prefix=/opt/install-awa
```

2.6 Build

After configuration is successful, you can build the library by running:

```
make
```

2.7 Installation

The installation is done by running the `install` target:

```
make install
```

2.8 Using

To use the library in an Ada project, add the following line at the beginning of your GNAT project file:

```
with "awa";
```

Depending on your application, you may also need to add the following GNAT projects which are provided by one or several of the libraries that Ada Web Application relies on:

```
with "utilada";
with "elada";
with "security";
with "servletada";
with "servletada_aws";
with "asf";
with "ado_mysql";
with "ado_sqlite";
with "ado_postgresql";
```

The library comes with several optional modules that you decide to use according to your needs. When you decide to use a module, you should add the GNAT project that corresponds to the module you wish to integrate For example, to use the `Jobs` and `Wikis` modules, you will need the following lines in your GNAT project:

```
with "awa_jobs";
with "awa_wikis";
```

3 Tutorial

Ada Web Application is a complete framework that allows to write web applications using the Ada language. Through a complete web application, the tutorial explains various aspects in setting up and building an application by using AWA.

The tutorial assumes that you have already installed the following software on your computer:

- The GNAT Ada compiler,
- The ArgoUML modelization tool,
- The Ada Web Application framework and its associated dependencies (XML/Ada and AWS),
- The Dynamo code generator.

3.1 The review web application

The review web application allows users to write reviews about a product, a software or a web site and share them to the Internet community. The community can read the review, participate by adding comments and voting for the reviewed product or software.

The AWA framework provides several modules that are ready to be used by our application. The login and user management is handled by the framework so this simplifies a lot the design of our application. We will see in the tutorial how we can leverage this to our review application.

Because users of our review web application have different roles, we will need permissions to make sure that only reviewers can modify a review. We will see how the AWA framework leverages the Ada Security library to enforce the permissions.

The AWA framework also integrates three other modules that we are going to use: the Tags Module, the Votes Module and the Comments Module.

Since many building blocks are already provided by the AWA framework, we will be able to concentrate on our own review application module.

3.2 Setting up the project

3.2.1 Project creation with Dynamo

The first step is to create the new project. Since creating a project from scratch is never easy we will use the Dynamo tool to build our initial review web application. Dynamo is a command line tool that provides several commands that help in several development tasks. For the project creation we will give:

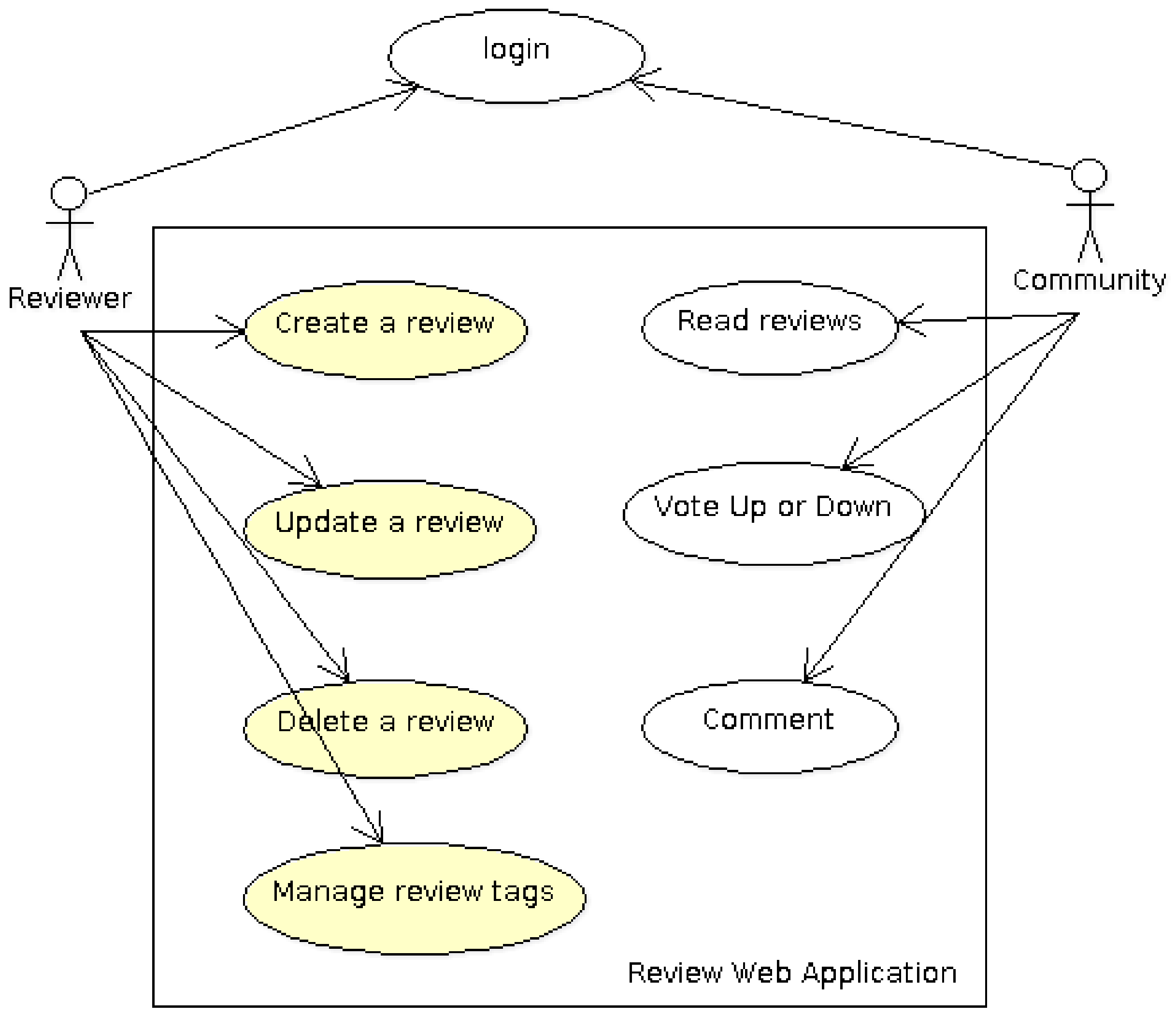


Figure 3: *Review Web Application Use Cases*

- the output directory,
- the project name,
- the license to be used for the project,
- the project author's email address.

Choose the project name with care as it defines the name of the Ada root package that will be used by the project. For the license, you have the choice between GPL v2, GPL v3, MIT, BSD 3 clauses, Apache 2 or some proprietary license.

```
dynamo -o atlas create-project -l apache atlas email@domain.com
```

The Dynamo project creation will build the `atlas` directory and populate it with many files:

- A set of configure, Makefile, GNAT project files to build the project,
- A set of Ada files to build your Ada web application,
- A set of presentation files for the web application.

Once the project is created, we must configure it to find the Ada compiler, libraries and so on. This is done by the following commands:

```
cd atlas
./configure
```

At this step, you may even build your new project and start it. The `make` command will build the Ada files and create the `bin/atlas-server` executable that represents the web application.

```
make generate build
bin/atlas-server start
```

Once the server is started, you may point your browser to the following location:

```
http://localhost:8080/atlas/index.html
```

3.2.2 Creating the review module with Dynamo

With the Ada Web Application framework, a web application is composed of modules where each module brings a specific functionality to the application. AWA provides a module for user management, another for comments, tags, votes, and many others. The application can decide to use these modules or not. The AWA module framework helps in defining the architecture and designing your web application.

For the review web application we will create our own module dedicated for the review management. The module will be an Ada child package of our root project package. From the Ada point of view, the final module will be composed of the following packages:

- A `Modules` package represents the business logic of the module. It is provides operations to access and manage the data owned by the module.
- A `Beans` package holds the Ada beans that make the link between the presentation layer and business logic.
- A `Models` package holds the data model to access the database content. This package is generated from UML and will be covered by a next tutorial.

To help in setting up a new AWA module, the Dynamo tool provides the `add-module` command. You just have to give the name of the module, which is the name of the Ada child package. Let's create our `reviews` module now:

```
dynamo add-module reviews
```

The command generates the new AWA module and modifies some existing files to register the new module in the application. You can build your web application at this stage even though the new module will not do anything yet for you.

3.3 Designing the data model

Our review web application will need to access a database to store the review information. For this, we must define a data model that will describe how the information is stored in the database and how we can access such information from Ada.

A Model Driven Engineering or MDE promotes the use of models to ease the development of software and systems. The Unified Modeling Language is used to modelize various parts of the software. UML is a graphical type modelling language and it has many diagrams but we are only going to use one of them: the Class Diagram.

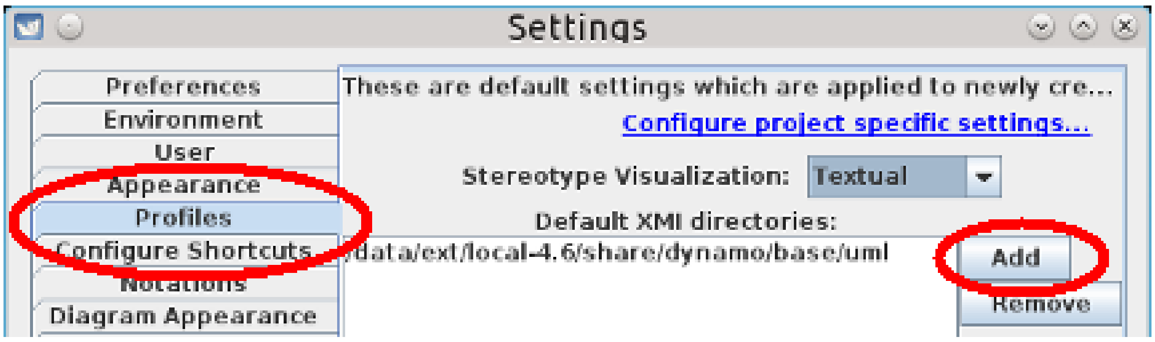
The class diagram is probably the most powerful diagram to design, explain and share the data model of any application. It defines the most important data types used by an application with the relation they have with each other. In the class diagram, a class represents an abstraction that encapsulates data member attributes and operations. The class may have relations with others classes.

3.3.1 ArgoUML setup

For the UML model, we are going to use ArgoUML that is a free modelization tool that works pretty well. For the ArgoUML setup, we will use two profiles:

- The Dynamo profile that describes the base data types for our UML model. These types are necessary for the code generator to work correctly.
- The AWA profile that describes the tables and modules provided by AWA. We will need it to get the user UML class definition.

These UML profiles are located in the `/usr/share/dynamo/base/uml` directory after Dynamo and AWA are installed. To configure ArgoUML, go in the `Edit -> Settings` menu and add the directory in the `Default XMI directories` list. Beware that you must restart ArgoUML to be able to use the new profiles.



These are default settings which are applied to newly cre...

Stereotype Visualization: Textual

Default XMI directories:
/data/ext/local-4.6/share/dynamo/base/uml

Figure 4: *Setting ArgoUML profiles*

Once the directory is added, restart ArgoUML, go again in `Edit -> Settings` menu and select the `AWA.xmi` and `Dynamo.xmi` profiles. As soon as they are selected and applied on the configuration, you should restart ArgoUML another time for these two profiles to become usable.

3.3.2 Modelize the domain model in UML

The UML model must use a number of Dynamo artifacts for the code generation to work properly. The artifact describes some capabilities and behavior for the code generator to perform its work. Stereotype names are enclosed within << and >> markers. Dynamo uses the following stereotypes:

- The `DataModel` stereotype must be applied on the package which contains the model to generate. This stereotype activates the code generation (other packages are not generated).
- The `Table` stereotype must be applied to the class. It controls which database table and Ada type will be generated.
- The `PK` stereotype must be defined in at most one attribute of the class. This indicates the primary key for the database table. The attribute type must be an integer or a string. This is a limitation of the Ada code generator.
- The `Version` stereotype must be applied on the attribute that is used for the optimistic locking implementation of the database layer.

- The `Auditable` stereotype can be applied to some attributes and relations when you want to audit changes to these attributes or relations. When used, the ADO framework will track changes and automatically record them in a specific auditing table.

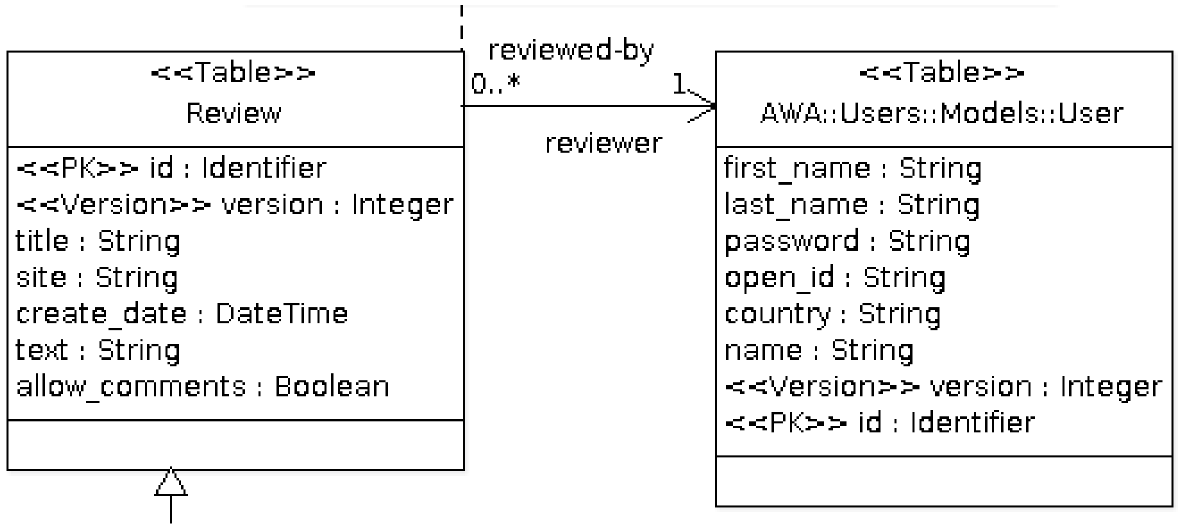


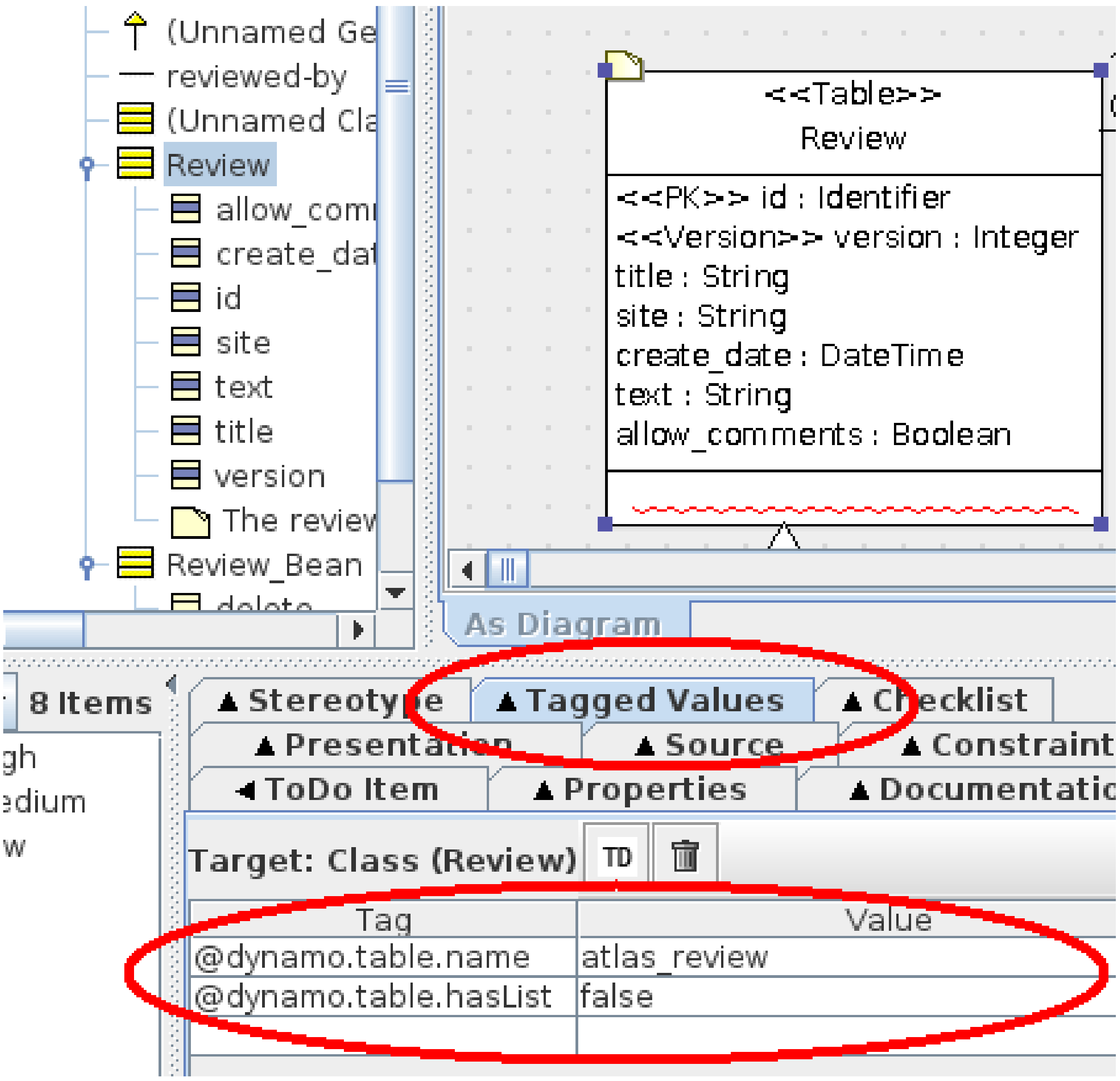
Figure 5: *The Review Table UML Model*

In our UML model, the `Review` table is assigned the `Table` stereotype so that an SQL table will be created as well as an Ada tagged type to represent our table. The `id` class attribute represents the primary key and thus has the `PK` stereotype. The `version` class attribute is the database column used by the optimistic locking implementation provided by Ada Database Objects. This is why is has the `Version` stereotype. The `title`, `site`, `create_date`, `text` and `allow_comments` attributes represent the information we want to store in the database table. They are general purpose attributes and thus don't need any specific stereotype. For each attribute, the Dynamo code generator will generate a getter and a setter operation that can be used in the Ada code.

To tune the generation, several UML tagged values can be selected and added on the table or on a table attribute. By applying a stereotype to the class, several tagged values can be added. By selecting the `Tagged Values` tab in ArgoUML we can edit and setup new values. For the `Review` table, the `dynamo.table.name` tagged value defines the name of the SQL database table, in our case `atlas_review`.

The `text` attribute in the `Review` table is a string that can hold some pretty long text. To control the length of the SQL column, we can set the `dynamo.sql.length` tagged value and tell what is that length.

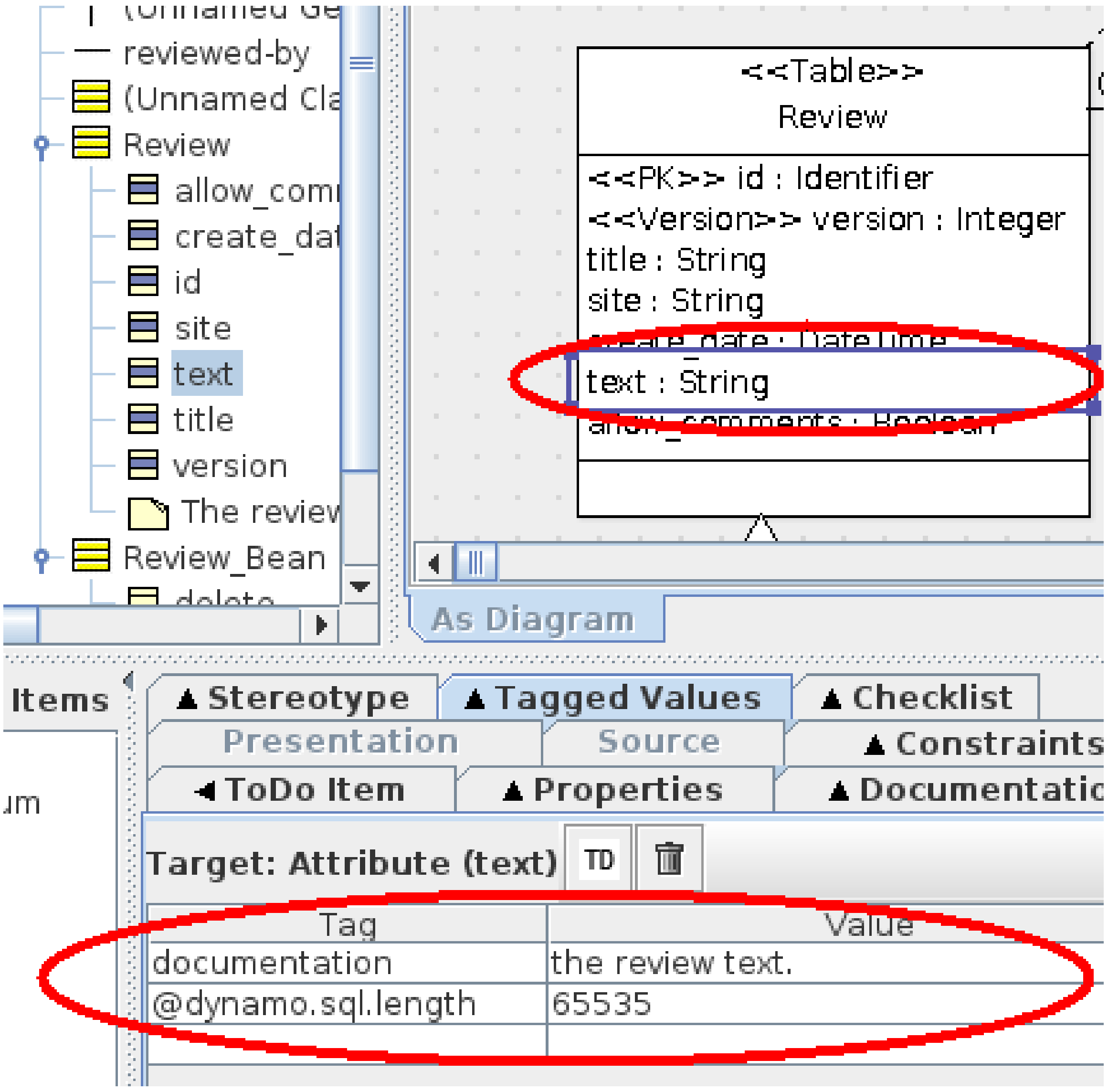
Once the UML model is designed, it is saved in the project directory `uml`. Dynamo will be able to read the ArgoUML file format (`.zargo` extension) so there is no need to export the UML in XMI.



Target: Class (Review)

Tag	Value
@dynamo.table.name	atlas_review
@dynamo.table.hasList	false

Figure 6: *The tagged value for the Review table*



Tag	Value
documentation	the review text.
@dynamo.sql.length	65535

Figure 7: *The tagged value for the text column in review table*

3.3.3 Adding relations in the UML model

The final UML model of our review application is fairly simple. We just added a table and a bean declaration. To benefit from the user management in AWA, we can use the `AWA::Users::Models::User` class that is defined in the AWA UML model. The `reviewed-by` association will create an attribute `reviewer` in our class. The code generator will generate a `Get_Reviewer` and `Set_Reviewer` operation in the Ada code. The SQL table will contain an additional column `reviewer` that will hold the primary key of the reviewer.

The `Review_Bean` class is an Ada Bean abstract class that will be generated by the code generator. The `Bean` stereotype activates the bean code generator and the generator will generate some code support that is necessary to turn the `Review_Bean` tagged record into an Ada Bean aware type. We will see in the section that we will only have to implement the `save` and `delete` operation that are described in this UML model.

3.3.4 Makefile setup

The `Makefile` that was generated by the Dynamo `create-project` command must be updated to setup a number of generation arguments for the UML to Ada code generator. Edit the `Makefile` to change `DYNAMO_ARGS` into:

```
DYNAMO_ARGS=--package Atlas.Reviews.Models db uml/atlas.zargo
```

The `--package` option tells Dynamo to generate only the model for the specified package. The `db` directory is the directory that will contain the SQL model files.

3.3.5 Generating the Ada model

To run the generator, we can use the `generate` make target:

```
make generate
```

The Dynamo code generator reads the file `uml/atlas.zargo` and the UML model it contains and generates:

- the Ada package `Atlas.Reviews.Models` which contains the definition of the `Review` table. The model files are created in the directory `src/models` which is separate from your Ada sources. You can safely remove the files in `src/models` and have them re-built by using Dynamo. It is not recommended to modify these files.

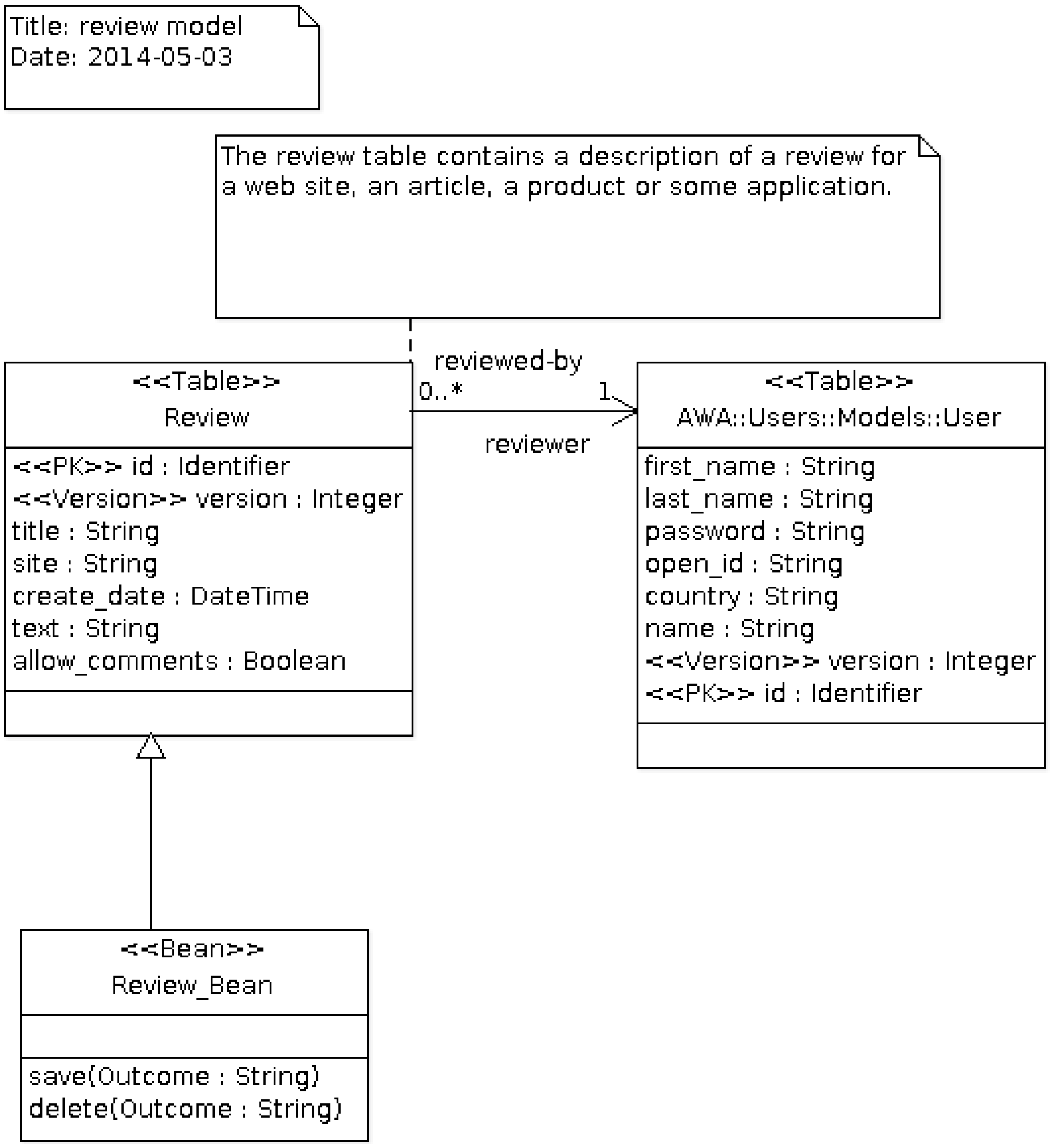


Figure 8: *The Review Web Application UML Model*

- the SQL files to create the MySQL or SQLite database. Depending on the AWA modules which are used, the generated SQL files will contain additional tables that are used by the AWA modules. The SQL files are generated in the db/mysql, db/sqlite and db/postgresql directories.

3.3.6 Creating the database

Until now we designed our application UML model, we have our Ada code generated, but we need a database with the tables for our application. We can do this by using the create-database command in Dynamo. This command needs several arguments:

- The directory that contains the SQL model files. In our case, this is db.
- The information to connect to the database, the database name, the user and its password. This information is passed in the form of a database connection string.
- The name of the database administration account to connect to the server and create the new database.
- The optional password for the database administration account.

If the MySQL server is running on your host and the admin account does not have any password, you can use the following command:

```
dynamo create-database \
    db 'mysql://localhost/demo_atlas?user=demo&password=demo' root
```

The create-database creates the database (demo_atlas) with the tables that are necessary for the application. It also creates the demo user and give it the necessary MySQL grants to connect to the demo_atlas database.

3.4 Adding a creation form

We will start with the presentation layer by adding two pages in our web application. A first page will contain the list of reviews and the second page will contain a form to create or update a review.

AWA uses the Facelets technology to allow developers write and design the presentation layer of the web application. This technology is commonly used in J2EE applications. A page is represented by an XML file that contains HTML code, includes some stylesheets, Javascript files and makes the link between the presentation and the web application.

3.4.1 Adding pages

Dynamo provides at least two commands that help in adding presentation files. The `add-page` command adds a simple page that can be edited and filled with real content. We will use it for the creation of the page to display the list of reviews.

```
dynamo add-page reviews/list
```

The `add-form` command creates another template of page that includes an HTML form to let a user submit some data to the web application.

```
dynamo add-form reviews/edit-review
```

These two commands will create the following files and they can now be modified.

```
./web/reviews/list.xhtml
./web/reviews/edit-review.xhtml
./web/reviews/forms/edit-review-form.xhtml
```

3.4.2 The create review form

In Facelets, an HTML form is created by using the `<h:form>` component from the HTML JSF namespace. This component will generate the HTML `form` tag and it will also manage the form submission.

The Ada Server Faces provides a set of widget components that facilitate the design of web application. The `<w:inputText>` component renders a title field with an HTML `<label>` and an HTML `<input>` text. We will use it to let the user enter the review title and the site URL being reviewed. The HTML `<textarea>` is provided by the JSF component `<h:inputTextArea>`. The review submit form is defined by the following XML extract:

```
<h:form xmlns:h="http://java.sun.com/jsf/html
  xmlns:w="http://code.google.com/p/ada-asf/widget">
  <h:inputHidden id='entity-id' value='#{review.id}' required='false'/>
  <w:inputText title='Title' value='#{review.title}'/>
  <w:inputText title='Site' value='#{review.site}'/>
  <h:inputTextArea rows='20' value='#{review.text}'/>
  <h:commandButton value='Save'
     action='#{review.save}'/>
</h:form>
```

Before closing the `<h:form>` component, we will put a `<h:commandButton>` that will render the form submit button.

3.4.3 How it works

Before going further, let's see how all this works. The principle below is exactly the same for a Java Server Faces application.

First, when the page is rendered the UEL expressions that it contains are evaluated. The `#{review.title}`, `#{review.site}` and `#{review.text}` are replaced by the content provided by the `review` object which is an Ada Bean provided by the `Review_Bean` tagged record.

When the page is submitted by the user, the input values submitted in the form are saved in the `review` bean, again by using the UEL expression. The `<h:commandButton>` action is then executed. This is also an UEL that indicates a method to invoke on the bean.

To sum up, the UEL makes the binding between the presentation layer in Facelets files and the Ada or Java beans.

The Ada Bean layer provides getter and setter to allow the UEL to retrieve and set values. For this, the `Review_Bean` tagged record implements two operations that are defined in the `[Bean](https://github.com/stcarrez/ada-util/source/browse/trunk/src/util-beans-basic.ads)` interface:

```
overriding
function Get_Value (From : in Review_Bean;
                    Name : in String) return Util.Beans.Objects.Object;

overriding
procedure Set_Value (From : in out Review_Bean;
                     Name : in String;
                     Value : in Util.Beans.Objects.Object);
```

The `Get_Value` operation is called to retrieve one of the Ada Bean member attribute and the `Set_Value` operation is called during form submission to set the member attribute.

Then the form button is pressed, the HTML form is submitted and received by the server. The `<h:form>` component identifies the form submission and each input component will validate the input fields. When everything has been validated, the `<h:commandButton>` component invokes the `Save` procedure that is declared as follows in the `Review_Bean` tagged record:

```
overriding
procedure Save (Bean : in out Review_Bean;
                Outcome : in out Ada.Strings.Unbounded.Unbounded_String
                   );
```

In the Ada Bean layer, we have to call the business logic to perform the `save` operation.

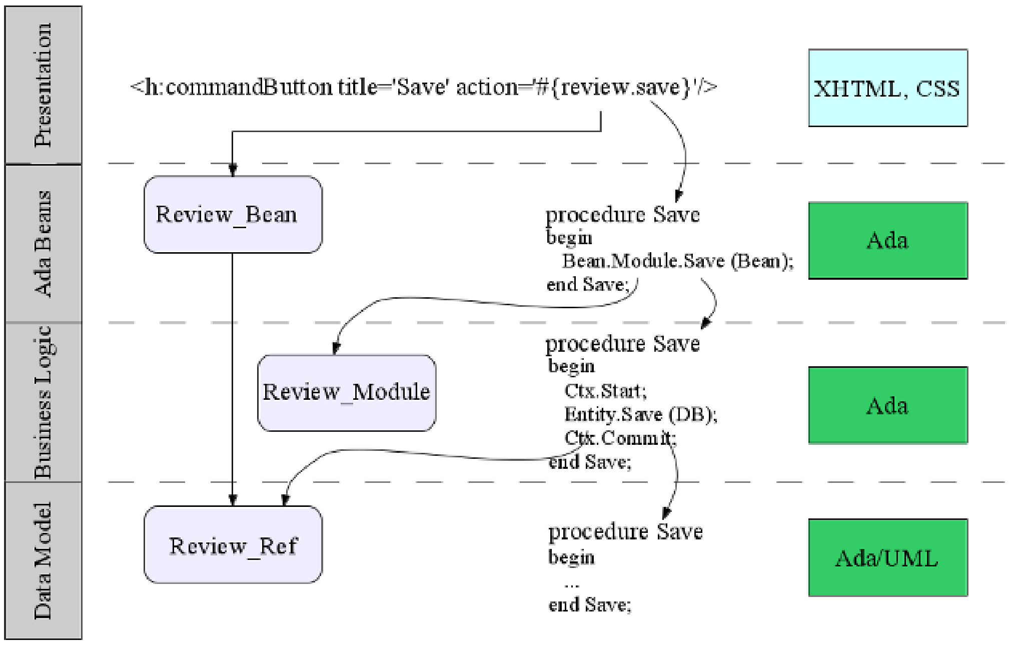


Figure 9: *Presentation, Ada Beans and Module interactions*

The business logic part is provided by the Ada module whose initial skeleton was generated by Dynamo. That layer is responsible for defining how the data is created, retrieved and modified. As far as we are concerned, this is rather simple since we only have to verify the permission and save the review object within some transaction. In other modules, several objects may be envolved and more complex rules may be defined for the integrity and validity of these objects.

The last part of the architecture is the data model layer that was in fact generated by Dynamo from the UML model. It is responsible for loading and saving Ada objects into the database.

3.4.4 The Review_Bean type declaration

When we designed our UML model, we have created the `Review_Bean` UML class and gave that class the `Bean` stereotype. We also declared two operations (`save` and `delete`) on that class. With this definition, Dynamo has generated in the `Atlas.Reviews.Models` package the `Review_Bean` abstract type. This type is abstract because we have to implement the `Save` and `Delete` operations. These are the two operations that can be called by an action such as used by the `<h:commandButton>` component.

The `Atlas.Reviews.Models` package is a generated package and it must not be modified. To implement our Ada Bean, we will add the `Review_Bean` type in our own package: the `Atlas.Reviews.Beans` package.

For this the `Review_Bean` type will inherit from the `Atlas.Reviews.Models.Review_Bean` type and it will implement the required operations. The type declaration looks like this:

```
package Atlas.Reviews.Beans is
...
type Review_Bean is new Atlas.Reviews.Models.Review_Bean with record
   Module : Atlas.Reviews.Modules.Review_Module_Access := null;
end record;
...
```

3.4.5 The Review_Bean implementation

The `Save` and `Delete` procedure must be implemented and since the whole business logic is managed by the module layer, we just have to call the associated module procedure as follows:

```
overriding
procedure Save (Bean : in out Review_Bean;
                Outcome : in out Ada.Strings.Unbounded.Unbounded_String
                   );
begin
   Bean.Module.Save (Bean);
end Save;

overriding
procedure Delete (Bean : in out Review_Bean;
                Outcome : in out Ada.Strings.Unbounded.Unbounded_String
                   );
begin
   Bean.Module.Delete (Bean);
end Delete;
```

3.4.6 The Review_Bean creation

The AWA framework must be able to create the `review` bean instance when a page is processed. For this, there are three steps that are necessary:

- we must define a create function whose role is to allocate the `Review_Bean` instance and return it. At the same time, the function can setup some pre-defined values for the object. The Dynamo tool has generated for us an example of such function so that there is nothing to do.

```
function Create_Review_Bean (Module : in Atlas.Reviews.Modules.
    Review_Module_Access)
   return Util.Beans.Basic.Readonly_Bean_Access is
   Object : constant Review_Bean_Access := new Review_Bean;
begin
   Object.Module := Module;
   return Object.all'Access;
end Create_Review_Bean;
```

- the creation function must be registered in the AWA framework under a name that identifies the create function. Again, an example of this registration has been generated by Dynamo and we are going to use it as is.

```
Register.Register (Plugin => Plugin,
                   Name    => "Atlas.Reviews.Beans.Reviews_Bean",
                   Handler => Atlas.Reviews.Beans.Create_Review_Bean'
                      Access);
```

- the last step is the configuration step. In the module XML configuration file, we must declare the Ada bean name and indicate what create function must be called to create it. We will use the managed-bean XML declaration that comes from Java Server Faces. We can declare as many Ada beans as we want each of them with a different name.

```
<managed-bean>
  <description>An example of a bean (change description and bean name)
     </description>
  <managed-bean-name>review</managed-bean-name>
  <managed-bean-class>Atlas.Reviews.Beans.Reviews_Bean</managed-bean-
     class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

When the UEL expression `#{review.title}` is used, the AWA framework looks for the Ada bean represented by `review` and identified by the `managed-bean-name` entry. It then calls the create function defined by the `managed-bean-class`. The Ada bean object is then stored either in the **request** context, a **session** context or an **application** context. This is defined by the `managed-bean-scope` entry. The **request** scope means that the Ada bean object is created once for each request. Concurrent page accesses will use their own Ada bean object instance. The **session** scope means that the Ada bean object is shared between requests on the same session. The **application** scope means that the Ada bean object is global to the application, shared by every request and every session.

3.4.7 Navigation rules

We have seen that when the review creation form is submitted the `<h:commandButton>` component has invoked the `Save` procedure of our `Review_Bean` object. The review object has been created and saved in the database and we kept the relation between the new review and the user.

We must now decide what should happen for the user to see the result. We could display a new form, update some page content or redirect to a new page. All this is defined by the navigation rules.

The navigation rules is the Java Server Faces mechanism that controls and defines what is the next page or view that must be displayed to a user. The navigation rules are configured in the module XML configuration file.

In the definition below, the navigation rule defines that the user is redirected to the page `/reviews/list.xhtml` if the current page was `/reviews/edit-review.xhtml` and the operation returned `success`.

```
<navigation-rule>
  <from-view-id>/reviews/edit-review.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/reviews/list.xhtml</to-view-id>
      <redirect/>
    </navigation-case>
</navigation-rule>
```

3.5 Creating the module

3.5.1 Adding the module operations

Now, we must add two operations on the business logic to save a review and delete a review. The Dynamo code generator provides the `add-module-operation` command that will help us in this task. Let's run it:

```
dynamo add-module-operation reviews review Save
dynamo add-module-operation reviews review Delete
```

The first parameter is the name of the module where the new operation is added. This is the name of the module that was created by using the `add-module` operation. In our case, this is the `reviews` module.

The second parameter is the name of the database entity or database table if you prefer.

The add-module-operation command modifies the Ada module specification and body to define and implement the following operation:

```ada
package Atlas.Reviews.Modules is
...
procedure Save (Model   : in Review_Module;
                Entity : in out Atlas.Reviews.Models.Review_Ref'Class);
...
```

The object to save in the Review table is passed as parameter to the Save operation. The procedure body that was generated is rather simple but functional: it just saves the object in the database within a transaction. In many cases it is ready to use but you may also need to modify the operation to either change the implementation or even add new parameters.

3.5.2 Saving our review

Before saving our review entity object, we want to associate it with the current user. We have to know who is the current user and for this we can use the AWA service context. The AWA service context is an object that is provided by the AWA.Services.Contexts package and that provides some useful contextual information for the business logic:

- It indicates the optional user that is authenticated and is doing the call,
- It gives access to the database connections that the business logic can use,
- It allows to manage database transactions.

The current service context is retrieved by using the AWA.Services.Contexts.Current function and we can use the Get_User function to know the current user. The Save procedure implementation is the following:

```ada
package ASC renames AWA.Services.Contexts;
procedure Save (Model   : in Review_Module;
                Entity : in out Atlas.Reviews.Models.Review_Ref'Class)
                   is
   Ctx    : constant ASC.Service_Context_Access := ASC.Current;
   DB     : ADO.Sessions.Master_Session := AWA.Services.Contexts.
      Get_Master_Session (Ctx);
begin
   Ctx.Start;
   if not Entity.Is_Inserted then
      Entity.Set_Reviewer (Ctx.Get_User);
      Entity.Set_Create_Date (Ada.Calendar.Clock);
   end if;
```

```
    Entity.Save (DB);
    Ctx.Commit;
end Save;
```

3.5.3 Setting up the permissions

Because we want to bring some minimal security to the review web application, we are going to setup some permissions that will be enforced by the business logic layer when a save or delete operation is done. The AWA framework uses the Ada Security to implement and enforce permissions. For this we need:

- An Ada definition of the permission,
- Adding a verification to enforce the permission in the new module operations,
- A definition of the permission rules.

Generating the permission

Dynamo provides the `add-permissions` command to help us in the first task. It generates some Ada code that declares the permissions. It also provides a default configuration for the new permissions.

```
dynamo add-permissions reviews review
```

The first parameter is the name of our module where the new permissions are declared and the second parameter is the name of the database entity. The command will modify the Ada module specification and add the following lines:

```
package Atlas.Reviews.Modules is
...
package ACL_Create_Reviews is new Security.Permissions.Definition ("
    review-create");
package ACL_Delete_Reviews is new Security.Permissions.Definition ("
    review-delete");
package ACL_Update_Reviews is new Security.Permissions.Definition ("
    review-update");
```

Each of these package instantiation, declares a single permission identified by a name.

Enforcing security

Now that we have our permission, we can enforce the security in the `Save` and `Delete` operation. This is done by using the `Check` operation provided by the `AWA.Permissions` package.

To verify that the user has the permission to create a new review, we can use the following call:

```
AWA.Permissions.Check (Permission => ACL_Create_Reviews.Permission);
```

This operation will verify that the user has the given permission and it will raise the `AWA.Permissions.NO_PERMISSION` exception if this is not the case. By raising such exception, the `Check` procedure acts as a barrier that grants or not the access to the rest of the code.

Now, if we have a review to modify, we will use the update permission and also give the review object to the `Check` operation so that it can verify if that particular review can be modified.

```
AWA.Permissions.Check (Permission => ACL_Update_Reviews.Permission,
                       Entity => Entity);
```

Configuring the permission

Until now we have created the permission and enforced it in the business logic. We have not defined the rules that tell what is really checked to verify the permission. The configuration part is defined in the XML file `config/reviews.xml` that was generated when the reviews module was created. The `add-permissions` command has modified the XML file to provide some default configuration. It has generated a XML permission for the `review-create`, `review-update` and `review-delete` permissions.

The `review-create` permission is defined as follows:

```
<auth-permission>
    <name>review-create</name>
</auth-permission>
```

This XML definition associate the Authenticated Permission controller to the `review-create` permission. With that controller the permission is granted if the security context has a principal (ie, a user is authenticated).

The `review-update` permission has another definition that we must change. Basically, we want that only the reviewer that created the review can update the review. For this we will use the entity permission controller provided by AWA. The XML definition is the following:

```
<entity-permission>
    <name>review-update</name>
    <entity-type>altas_review</entity-type>
    <sql>
       SELECT r.id FROM atlas_review AS r
       WHERE r.id = :entity_id AND r.reviewer_id = :user_id
    </sql>
</entity-permission>
```

When the permission is checked, the entity permission controller will use the SQL statement to verify the permission. The SQL statement has three parameters:

- `user_id` is the ID of the user associated with the security context. If there is no authentified user, the permission is refused.
- `entity_id` is the ID of the database entity as passed to the `Check` procedure and propagated to the permission controller.
- `entity_type` is a unique number that identifies the database entity type or database table if you prefer. It is created and setup automatically according to the entity type defined in the `entity-type` XML member. It is not used in our example.

At the end, the above SQL statement verifies that the review exists and was created by the current user.

3.6 Using database queries

Our next step is now to list the reviews that have been created. We need to add a page that will list the reviews and we need to implement a database query to fetch the information.

3.6.1 Adding database queries

Since we need to access the list of reviews from the XHTML files, we will map the SQL query result to a list of Ada Beans objects. For this, an [XML query mapping|https://code.google.com/p/ada-ado/wiki/QueryMapping] is created to tell how to map the SQL query result into some Ada record. The XML query mapping is then processed by Dynamo to generate the Ada Beans implementation. The XML query mapping is also read by AWA to get the SQL query to execute.

A template of the XML query mapping can be added to a project by using the dynamo `add-query` command. The first parameter is the module name (`reviews`) and the second parameter the name of the query (`list`). The command will generate the file `db/reviews-list.xml`.

```
dynamo add-query reviews list
```

The generated XML query mapping is an example of a query. You can replace it or update it according to your needs. The first part of the XML query mapping is a **`class`** declaration that describes the type to represent each row returned by our query. Within the **`class`**, a set of `property` definition describes the class attributes with their type and name.

```
<query-mapping package='Atlas.Reviews.Models'>
    <class name="Atlas.Reviews.Models.List_Info" bean="yes">
        <comment>The list of reviews.</comment>
        <property type='Identifier' name="id">
            <comment>the review identifier.</comment>
        </property>
        <property type='String' name="title">
            <comment>the review title.</comment>
        </property>
        ...
    </class>
</query-mapping>
```

Following the **class** declaration, the `query` declaration describes a query by giving it a name and describing the SQL statement to execute. By having the SQL statement separate and external to the application, we can update, fix and tune the SQL without rebuilding the application. The Dynamo code generator will use the `query` declaration to generate a query definition that can be referenced and used from the Ada code.

The SQL statement is defined within the `sql` XML entity. The optional `sql-count` XML entity is used to associate a count query that can be used for the pagination.

We want to display the review with the author's name and email address. The list will be sorted by date to show the newest reviews first. The SQL to execute is the following:

```
<query-mapping package='Atlas.Reviews.Models'>
   ...
    <query name='list'>
       <comment>Get the list of reviews</comment>
       <sql>
SELECT
      r.id,
      r.title,
      r.site,
      r.create_date,
      r.allow_comments,
      r.reviewer_id,
      a.name,
      e.email,
      r.text
FROM atlas_review AS r
INNER JOIN awa_user AS a ON r.reviewer_id = a.id
INNER JOIN awa_email AS e ON a.email_id = e.id
ORDER BY r.create_date DESC
    LIMIT :first, :last
       </sql>
       <sql-count>
    SELECT
      count(r.id)
    FROM atlas_review AS r
       </sql-count>
    </query>
</query-mapping>
```

The query has two named parameters represented by `:first` and `:last`. These parameters allow to paginate the list of reviews.

The complete source can be seen in the file: db/reviews-list.xml.

Once the XML query is written, the Ada code is generated by Dynamo by reading the UML model and all the XML query mapping defined for the application. Dynamo merges all the definitions into the target Ada packages and generates the Ada code in the `src/model` directory. You can use the `generate` make target:

```
make generate
```

or run the following command manually:

```
dynamo generate db uml/atlas.zargo
```

From the `List_Info` class definition, Dynamo generates the `List_Info` tagged record. The record contains all the data members described in the **`class`** XML entity description. The `List_Info` represents one row returned by the SQL query. The attributes of the `List_Info` can be accessed from the XHTML files by using UEL expression and the property name defined for each attribute.

To describe the list of rows, Dynamo generates the `List_Info_Beans` package which instantiates the `Util.Beans.Basic.Lists` generic package. This provides an Ada vector for the `List_Info` type and an Ada bean that gives access to the list.

```
package Atlas.Reviews.Models is
  ...
  type List_Info is new Util.Beans.Basic.Readonly_Bean with record
  ...
   package List_Info_Beans is
      new Util.Beans.Basic.Lists (Element_Type => List_Info);
   package List_Info_Vectors renames List_Info_Beans.Vectors;
   subtype List_Info_List_Bean is List_Info_Beans.List_Bean;
   subtype List_Info_Vector is List_Info_Vectors.Vector;
   Query_List : constant ADO.Queries.Query_Definition_Access;
   ...
end Atlas.Reviews.Models;
```

The generated code can be seen in src/model/atlas-reviews-models.ads.

3.6.2 Implementing the review list bean

In order to access the list of reviews from the XHTML facelet file, we must create an Ada bean that provides the list of reviews. This Ada bean is modelized in the UML model and we define:

- A set of attributes to manage the review list pagination (`page`, `page_size`, `count`)
- An Ada bean action that can be called from the XHTML facelet file (`load`)

The `Review_List_Bean` tagged record will hold the list of reviews for us:

```
package Atlas.Reviews.Beans is
  ...
   type Review_List_Bean is new Atlas.Reviews.Models.Review_List_Bean
      with record
      Module       : Atlas.Reviews.Modules.Review_Module_Access := null
         ;
      Reviews      : aliased Atlas.Reviews.Models.List_Info_List_Bean;
      Reviews_Bean : Atlas.Reviews.Models.List_Info_List_Bean_Access;
   end record;
   type Review_List_Bean_Access is access all Review_List_Bean'Class;
end Atlas.Reviews.Beans;
```

We must now implement the `Load` operation that was described in the UML model and we are going to use our `list` query. For this, we use the `ADO.Queries.Context` to setup the query to retrieve the list of reviews. A call to `Set_Query` indicates the query that will be used. Since that query needs two parameters (`first` and `last`), we use the `Bind_Param` operation to give the two values. The list of reviews is then retrieved easily by calling the `Atlas.Reviews.Models.List` operation that was generated by Dynamo.

```
package body Atlas.Reviews.Beans is
...
   overriding
   procedure Load (Into     : in out Review_List_Bean;
                    Outcome : in out Ada.Strings.Unbounded.
                       Unbounded_String) is
      Session     : ADO.Sessions.Session := Into.Module.Get_Session;
      Query       : ADO.Queries.Context;
      Count_Query : ADO.Queries.Context;
      First       : constant Natural  := (Into.Page - 1) * Into.
         Page_Size;
      Last        : constant Positive := First + Into.Page_Size;
   begin
      Query.Set_Query (Atlas.Reviews.Models.Query_List);
      Count_Query.Set_Count_Query (Atlas.Reviews.Models.Query_List);
      Query.Bind_Param (Name => "first", Value => First);
      Query.Bind_Param (Name => "last", Value => Last);
      Atlas.Reviews.Models.List (Into.Reviews, Session, Query);
      Into.Count := ADO.Datasets.Get_Count (Session, Count_Query);
   end Load;
end Atlas.Reviews.Beans;
```

3.6.3 Review list bean creation

The AWA framework must be able to create an instance of the `Review_List_Bean` type. For this, we have to declare and implement a constructor function that allocates an instance of the `Review_List_Bean` type and setup some pre-defined values. When the instance is returned, the list of reviews is not loaded.

```
package body Atlas.Reviews.Beans is
   ...
   function Create_Review_List_Bean (Module : in Atlas.Reviews.Modules.
      Review_Module_Access)
                                  return Util.Beans.Basic.
                                     Readonly_Bean_Access is
      Object   : constant Review_List_Bean_Access := new
         Review_List_Bean;
   begin
      Object.Module       := Module;
      Object.Reviews_Bean := Object.Reviews'Access;
      Object.Page_Size    := 20;
      Object.Page         := 1;
      Object.Count        := 0;
      return Object.all'Access;
   end Create_Review_List_Bean;
end Atlas.Reviews.Beans;
```

The constructor function is then registered in the `Atlas.Reviews.Modules` package within the `Initialize` procedure. This registration allows to give a name for this constructor function and be able to specify it in the `managed-bean` bean declaration.

```
package body Atlas.Reviews.Modules is
   ...
   overriding
   procedure Initialize (Plugin : in out Review_Module;
                         App    : in AWA.Modules.Application_Access;
                         Props  : in ASF.Applications.Config) is
   begin
      ...
      Register.Register (Plugin => Plugin,
                         Name   => "Atlas.Reviews.Beans.
                            Review_List_Bean",
                         Handler => Atlas.Reviews.Beans.
                            Create_Review_List_Bean'Access);
   end Initialize;
end Atlas.Reviews.Modules;
```

3.6.4 Review list bean declaration

The managed-bean XML declaration associates a name to a constructor function that will be called when the name is needed. The scope of the Ada bean is set to `request` so that a new instance is created for each HTTP GET request.

```
<managed-bean>
  <description>The list of reviews</description>
  <managed-bean-name>reviewList</managed-bean-name>
  <managed-bean-class>Atlas.Reviews.Beans.Review_List_Bean</managed-
      bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Two other scopes are allowed: `session` and `application`. The `session` scope indicates that the new instance is created and associated with the user browsing session. It allows to share some instance between several HTTP requests. Care must be made when designing the Ada bean instance because concurrent HTTP requests can access and modify the Ada bean concurrently.

The `application` scope associates the new instance globally to the application. It means the instance is shared across all requests concurrently.

3.6.5 Listing the reviews: the XHTML facelet presentation file

To load the reviews to be displayed we will use a JSF 2.2 view action. The review list page has a parameter `page` that indicates the page number to be displayed. The `f:viewParam` allows to retrieve that parameter and configure the `reviewList` Ada bean with it. Then, the `f:viewAction` defines the action that will be executed after the view parameters are extracted, validated and passed to the Ada bean. In our case, we will call the `load` operation on our `reviewList` Ada bean.

```
<f:metadata>
    <f:viewParam id='page' value='#{reviewList.page}' required="false"
       />
    <f:viewAction action="#{reviewList.load}"/>
</f:metadata>
```

To summarize, the `reviewList` Ada bean is created, then configured for the pagination and filled with the current page content by running our SQL query by running the `Load` procedure.

The easy part is now to render the list of reviews. The XHTML file uses the component to iterate over the list items and render each of them. At each iteration, the `<h:list>` component initializes the Ada bean `review` to refer to the current row in the review list. We can then access each attribute defined in the XML query mapping by using the property name of that attribute. For example `review.title` returns the `title` property.

```
<h:list var="review" value="#{reviewList.reviews}">
    <div class='review' id="p_#{review.id}">
        <div class='review-title'>
            <h2><a href="#{review.site}">#{review.title}</a></h2>
            <ul class='review-info'>
                <li><span>By #{review.reviewer_name}</span></li>
                <li>
                    <h:outputText styleClass='review-date'
                                  value="#{review.date}"
                                  converter="dateConverter"/>
                </li>
                <h:panelGroup rendered="#{review.reviewer_id == user.id
                  }">
                    <li>
                        <a href="#{contextPath}/reviews/edit-review.
                           html?id=#{review.id}">
```

```
                            #{reviewMsg.review_edit_label}
                        </a>
                    </li>
                    <li>
                        <a href="#"
                            onclick="return ASF.OpenDialog(this, 'deleteDialog', '#{contextPath}/reviews/forms/delete-review.html?id=#{review.id}');">
                            #{reviewMsg.review_delete_label}
                        </a>
                    </li>
                </h:panelGroup>
            </ul>
        </div>
        <awa:wiki styleClass='review-text post-text' value="#{review.text}" format="dotclear"/>
    </div>
</h:list>
```

3.6.6 Understanding the request flow

Let's see the whole request flow to better understand what happens.

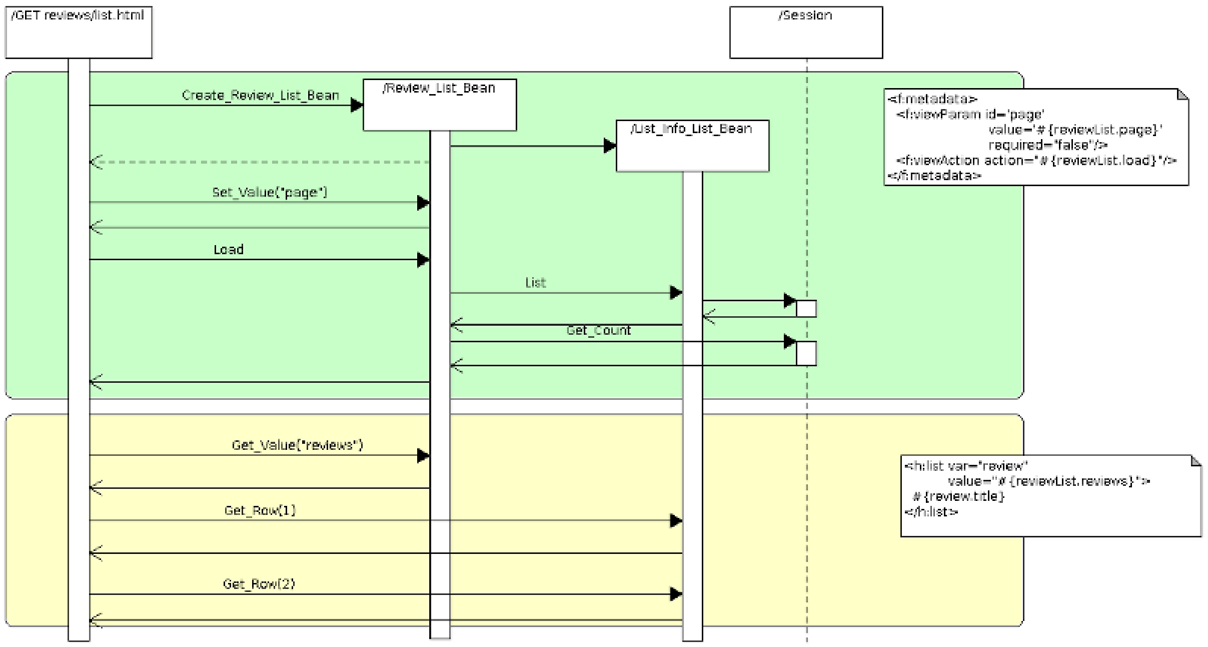
To display the list of reviews, the user's browser makes an HTTP GET request to the page `/reviews/list.html`. This page maps to the XHTML file `web/reviews/list.xhtml` that we created in the Adding pages section.

The Ada Server Faces framework handles the request by first reading the XHTML file and building a tree of components that represent the view to render. Within that tree of component, the `<f:metadata>` component allows to make a pre-initialization of components and Ada beans before the component tree is rendered.

For the pre-initialization, the `reviewList` Ada bean is created because it is referenced in an EL expression used by the `<f:viewParam>` component or by the `<f:viewAction>`. For this creation, the `Create_Review_List_Bean` constructor that we registered is called. The `page` attribute is set on the `reviewList` Ada bean if it was passed as a URL request parameter.

The `load` action is then called by Ada Server Faces which triggers execution of the `Load` procedure and the current review list page is retrieved by executing the SQL query.

As soon as the `load` action terminates, the rendering of the component tree can be processed. The `reviewList` Ada bean contains the information to display and the `<h:list>` component iterates over the list and renders each row at a time.



```
<f:metadata>
  <f:viewParam id='page'
               value='#{reviewList.page}'
               required="false"/>
  <f:viewAction action="#{reviewList.load}"/>
</f:metadata>
```

```
<h:list var="review"
        value="#{reviewList.reviews}">
  #{review.title}
</h:list>
```

Figure 10: *Review list flow*

4 AWA Core

4.1 Initialization

The AWA application is represented by the `Application` type which should be extended for the final application to provide the modules and specific components of the final application.

The initialization of an AWA application is made in several steps represented by different procedures of the main `Application` type. The whole initialization is handled by the `Initialize` procedure which gets a first set of configuration properties and a factory to build specific component.

The `Initialize` procedure will perform the following steps:

- It uses the factory to allocate the ASF lifecycle handler, the navigation handler, the security manager, the OAuth manager, the exception handlers.
- It calls the `Initialize_Components` procedure to let the application register all the ASF components. These components must be registered before any configuration file is read.
- It calls the `Initialize_Config`
- It calls the `Initialize_Servlets` procedure to allow the application to register all the servlet instances used by the application.
- It calls the `Initialize_Filters` procedure to allow the application to register all the servlet filter instances. The application servlets and filters must be registered before reading the global configuration file.
- It loads the global application configuration by reading the `awa.xml` file. By reading this configuration, some global configuration is established on the servlets, filters.
- It calls the `Initialize_Modules` procedure so that all the application modules can be registered, configured and initialized. Each module brings its own component, servlet and filter. They are configured by their own XML configuration file.
- It loads the module application configuration by reading the XML files described by the `app.config.plugins` configuration. This last step allows the application to setup and update the configuration of all modules that have been registered.

4.2 Configuration

The following global configuration parameter are defined:

Name	Description
awa_url_scheme	The application URL scheme to use when building URL.
	#{empty app_url_scheme ? "http://" : app_url_scheme}
awa_url_host	The application URL host to use when building URL.
	#{empty app_url_host ? "localhost" : app_url_host}
awa_url_port	The application TCP port to use when building URL.
	#{empty app_url_port ? ":8080" : app_url_port}
app_url_base	The application URL base to use when building URL.
	#{empty app_url_base ? "http://localhost:8080" : app_url_base}
app_oauth_url_base	
	http://localhost:8080
view.ext	Defines the extension used for Ada Server Faces presentation pages.
	.html
view.dir	Defines a list of paths separated by ";" where the XHTML files are searched. The default searches for the "web" directory in the application search paths.
	#{fn:composePath(app_search_dirs,"web")}
content-type.default	Defines the default content type for the file servlet.
	text/plain
ado.queries.load	Controls whether the database query definitions are loaded.
	true
ado.queries.paths	Defines a list of paths separated by ";" where the database query files are searched. The default searches for the "db" directory in the application search paths.
	#{fn:composePath(app_search_dirs,"db")}
bundle.dir	Defines a list of paths separated by ";" where the resource bundle files are searched. The default searches for the "bundles" directory in the application search paths.
	#{fn:composePath(app_search_dirs,"bundles")}

Name	Description
app.modules.dir	Defines a list of paths separated by ";" where the module configuration files are searched. The default searches for the "config" directory in the application search paths.
	#{fn:composePath(app_search_dirs,"config")}

4.3 AWA Modules

A module is a software component that can be integrated in the web application. The module can bring a set of service APIs, some Ada beans and some presentation files. The AWA framework allows to configure various parts of a module when it is integrated in an application. Some modules are designed to be re-used by several applications (for example a *mail* module, a *users* module, …). Other modules could be specific to an application. An application will be made of several modules, some will be generic some others specific to the application.

4.3.1 Registration

The module should have only one instance per application and it must be registered when the application is initialized. The module instance should be added to the application record as follows:

```ada
type Application is new AWA.Applications.Application with record
   Xxx       : aliased Xxx_Module;
end record;
```

The application record must override the `Initialize_Module` procedure and it must register the module instance. This is done as follows:

```ada
overriding
procedure Initialize_Modules (App : in out Application) is
begin
   Register (App    => App.Self.all'Access,
             Name   => Xxx.Module.NAME,
             URI    => "xxx",
             Module => App.User_Module'Access);
end Initialize_Modules;
```

The module is registered under a unique name. That name is used to load the module configuration.

4.3.2 Configuration

The module is configured by using an XML or a properties file. The configuration file is used to define:

- the Ada beans that the module defines and uses,
- the events that the module wants to receive and the action that must be performed when the event is posted,
- the permissions that the module needs and how to check them,
- the navigation rules which are used for the module web interface,
- the servlet and filter mappings used by the module

The module configuration is located in the *config* directory and must be the name of the module followed by the file extension (example: `module-name`.xml or `module-name`.properties).

4.4 AWA Permissions

The *AWA.Permissions* framework defines and controls the permissions used by an application to verify and grant access to the data and application service. The framework provides a set of services and API that helps an application in enforcing its specific permissions. Permissions are verified by a permission controller which uses the service context to have information about the user and other context. The framework allows to use different kinds of permission controllers. The `Entity_Controller` is the default permission controller which uses the database and an XML configuration to verify a permission.

4.4.1 Declaration

To be used in the application, the first step is to declare the permission. This is a static definition of the permission that will be used to ask to verify the permission. The permission is given a unique name that will be used in configuration files:

```ada
with Security.Permissions;
...
package ACL_Create_Post is new Security.Permissions.Definition ("blog-
    create-post");
```

4.4.2 Checking for a permission

A permission can be checked in Ada as well as in the presentation pages. This is done by using the `Check` procedure and the permission definition. This operation acts as a barrier: it does not return anything but returns normally if the permission is granted. If the permission is denied, it raises the `NO_PERMISSION` exception.

Several `Check` operation exists. Some require no argument and some others need a context such as some entity identifier to perform the check.

```
with AWA.Permissions;
...
AWA.Permissions.Check (Permission => ACL_Create_Post.Permission,
                       Entity     => Blog_Id);
```

4.4.3 Configuring a permission

The *AWA.Permissions* framework supports a simple permission model The application configuration file must provide some information to help in checking the permission. The permission name is referenced by the `name` XML entity. The `entity-type` refers to the database entity (ie, the table) that the permission concerns. The `sql` XML entity represents the SQL statement that must be used to verify the permission.

```
<entity-permission>
  <name>blog-create-post</name>
  <entity-type>blog</entity-type>
  <description>Permission to create a new post.</description>
  <sql>
    SELECT acl.id FROM acl
    WHERE acl.entity_type = :entity_type
    AND acl.user_id = :user_id
    AND acl.entity_id = :entity_id
  </sql>
</entity-permission>
```

4.4.4 Adding a permission

Adding a permission means to create an `ACL` database record that links a given database entity to the user. This is done easily with the `Add_Permission` procedure:

```
with AWA.Permissions.Services;
...
AWA.Permissions.Services.Add_Permission (Session => DB,
                                         User    => User,
                                     Entity  => Blog);
```

4.4.5 Data Model

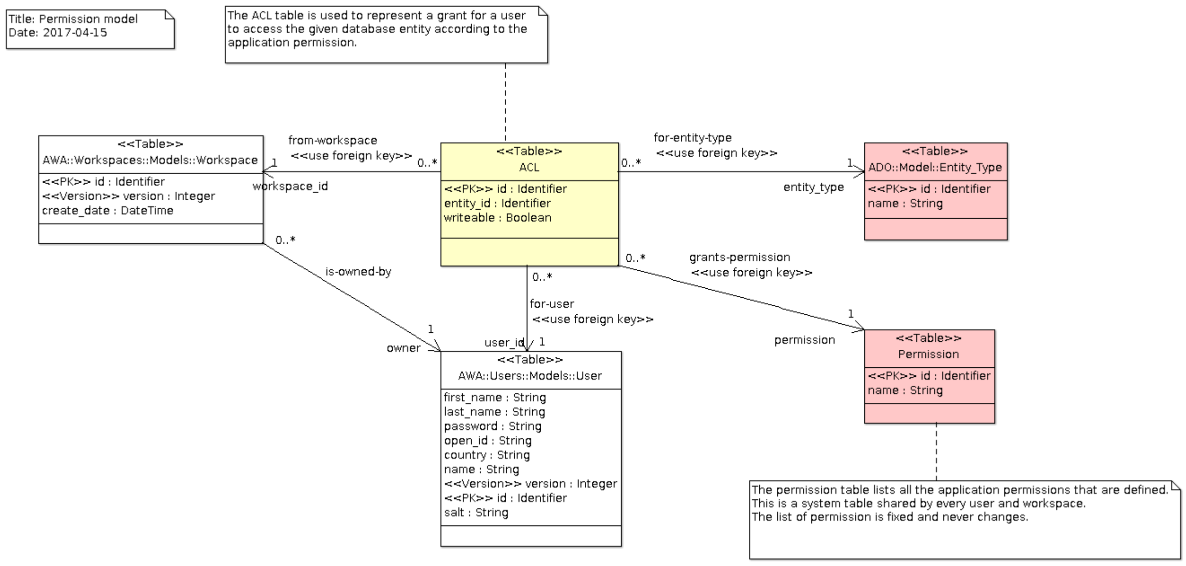


Figure 11

4.4.6 Queries

4.4.7 Queries

Name	Description
check-entity-permission	Get the permission for a user and an entity
remove-permission	Delete the permission associated with a user and an object
remove-entity-permission	Delete all the permission associated with an object
remove-user-permission	Delete all the permission associated with a user

Name	Description

4.5 AWA Events

The AWA.Events package defines an event framework for modules to post events and have Ada bean methods be invoked when these events are dispatched. Subscription to events is done through configuration files. This allows to configure the modules and integrate them together easily at configuration time.

4.5.1 Declaration

Modules define the events that they can generate by instantiating the Definition package. This is a static definition of the event. Each event is given a unique name.

```
with AWA.Events.Definition;
...
package Event_New_User is new AWA.Events.Definition ("new-user");
```

4.5.2 Posting an event

The module can post an event to inform other modules or the system that a particular action occurred. The module creates the event instance of type Module_Event and populates that event with useful properties for event receivers.

```
with AWA.Events;
...
Event : AWA.Events.Module_Event;
Event.Set_Event_Kind (Event_New_User.Kind);
Event.Set_Parameter ("email", "harry.potter@hogwarts.org");
```

The module will post the event by using the Send_Event operation.

```
Manager.Send_Event (Event);
```

4.5.3 Receiving an event

Modules or applications interested by a particular event will configure the event manager to dispatch the event to an Ada bean event action. The Ada bean is an object that must implement a procedure that matches the prototype:

```ada
type Action_Bean is new Util.Beans.Basic.Readonly_Bean ...;
procedure Action (Bean : in out Action_Bean;
                  Event : in AWA.Events.Module_Event'Class);
```

The Ada bean method and object are registered as other Ada beans.

The configuration file indicates how to bind the Ada bean action and the event together. The action is specified using an EL Method Expression (See Ada EL or JSR 245).

```xml
<on-event name="new_user">
    <action>#{ada_bean.action}</action>
</on-event>
```

4.5.4 Event queues and dispatchers

The `AWA.Events` framework posts events on queues and it uses a dispatcher to process them. There are two kinds of dispatchers:

- Synchronous dispatcher process the event when it is posted. The task which posts the event invokes the Ada bean action. In this dispatching mode, there is no event queue. If the action method raises an exception, it will however be blocked.
- Asynchronous dispatcher are executed by dedicated tasks. The event is put in an event queue. A dispatcher task processes the event and invokes the action method at a later time.

When the event is queued, there are two types of event queues:

- A Fifo memory queue manages the event and dispatches them in FIFO order. If the application is stopped, the events present in the Fifo queue are lost.
- A persistent event queue manages the event in a similar way as the FIFO queue but saves them in the database. If the application is stopped, events that have not yet been processed will be dispatched when the application is started again.

4.5.5 Data Model

4.6 AWA Commands

The `AWA.Commands` package provides a simple framework with commands that allow to start, stop, configure and manage the web application. It is also possible to provide your own commands. The

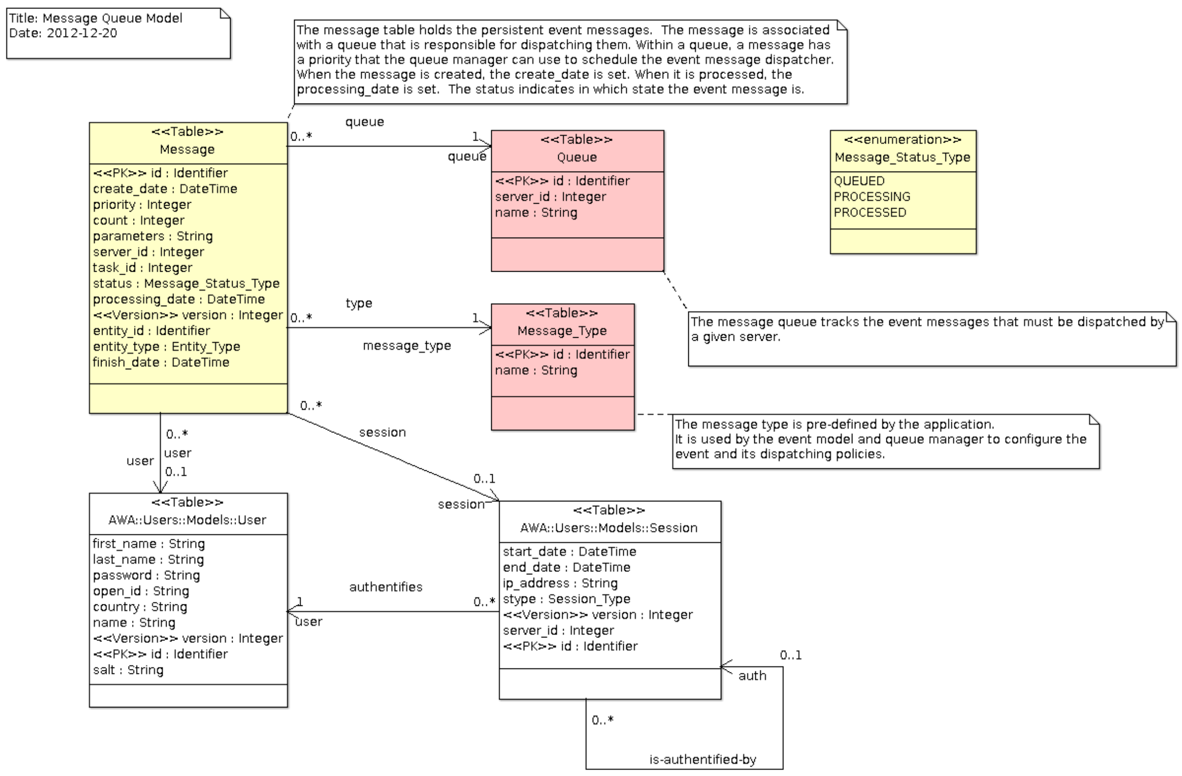

Title: Message Queue Model
Date: 2012-12-20
The message table holds the persistent event messages. The message is associated with a queue that is responsible for dispatching them. Within a queue, a message has a priority that the queue manager can use to schedule the event message dispatcher. When the message is created, the create_date is set. When it is processed, the processing_date is set. The status indicates in which state the event message is.
<<Table>>
Message
<<PK>> id : Identifier
create_date : DateTime
priority : Integer
count : Integer
parameters : String
server_id : Integer
task_id : Integer
status : Message_Status_Type
processing_date : DateTime
<<Version>> version : Integer
entity_id : Identifier
entity_type : Entity_Type
finish_date : DateTime
queue
0..*
1
queue
<<Table>>
Queue
<<PK>> id : Identifier
server_id : Integer
name : String
<<enumeration>>
Message_Status_Type
QUEUED
PROCESSING
PROCESSED
The message queue tracks the event messages that must be dispatched by a given server.
type
0..*
1
message_type
<<Table>>
Message_Type
<<PK>> id : Identifier
name : String
The message type is pre-defined by the application.
It is used by the event model and queue manager to configure the event and its dispatching policies.
0..*
session
0..1
session
0..*
user
user
0..1
<<Table>>
AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
authentifies
1
user
0..*
<<Table>>
AWA::Users::Models::Session
start_date : DateTime
end_date : DateTime
ip_address : String
stype : Session_Type
<<Version>> version : Integer
server_id : Integer
<<PK>> id : Identifier
0..1
auth
0..*
is-authentified-by

Figure 12

command framework handles the parsing of command line options, identification of the command to execute and execution of the selected command.

4.6.1 Command Usage

SYNOPSIS

driver [-v] [-vv] [-vvv] [-c *config-file*] *command* [-k *file*] [-d *dir*] [-p *password*] [–password *password*] [–passfile *file*] [–passenv *name*] [–passfd *fd*] [–passask] [–passcmd *cmd*] [–wallet-key-file *file*]

DESCRIPTION

The `AWA.Commands.Drivers` framework integrates the Ada Keystore support to access some sensitive configuration information such as passwords, database connection strings, API secret keys. The use of the Ada Keystore storage is optional. It is enabled when the `-k _file_` option is specified. When such secure storage is used, a primary password to unlock the keystore is necessary. Passwords are retrieved using one of the following options:

- by reading a file that contains the password,
- by looking at an environment variable,
- by using a command line argument,
- by getting the password through the *ssh-askpass*(1) external command,
- by running an external command,
- by using a GPG private key,
- by asking interactively the user for the password,
- by asking through a network socket for the password.

When the Ada Keystore is used, it is global to all the applications that are registered in the Web server container. To allow to differentiate application specific configuration, each configuration parameter is prefixed by the application name.

To create and update the keystore file, it is necessary to use the *akt*(1) tool. The tool provides many commands for the creation, insertion, removal and update of content that is stored in the keystore file.

If the keystore file was locked by using GPG, it is not necessary to specify any specific option to unlock the keystore. All is needed is the availability of the *gpg2*(1) command with the private key to unlock the keystore.

The server global configuration file that is read with the `-c config_file` option can contain the following configuration:

Name	Description
keystore-path	The path to the global keystore file
keystore-masterkey-path	The path of the file that contains the master keys
keystore-password-path	The path of the file that contains the keystore password
gpg-encrypt	When GPG is used, the GPG command to encrypt some content
gpg-decrypt	When GPG is used, the GPG command to decrypt some content
gpg-list-keys	When GPG is used, the GPG command to list the available GPG keys

OPTIONS

The following options are recognized by the command driver:

-V

Prints the application version.

-v

Enable the verbose mode.

-vv

Enable debugging output.

-vvv

Enable debugging output.

-c *config-file*

Defines the path of the global server configuration file.

-k file

Specifies the path of the keystore file to open.

-p password

The keystore password is passed within the command line. Using this method is convenient but is not safe.

–passenv envname

The keystore password is passed within an environment variable with the given name. Using this method is considered safer but still provides some security leaks.

–passfile path

The keystore password is passed within a file that is read. The file must not be readable or writable by other users or group: its mode must be r??——. The directory that contains the file must also satisfy the not readable by other users or group members, This method is safer.

–passfd fd

The keystore password is passed within a pipe whose file descriptor number is given. The file descriptor is read to obtain the password. This method is safer.

–passask

The keystore password is retrieved by the running the external tool *ssh-askpass*(1) which will ask the password through either KDE, Gnome or another desktop interactive application. The password is retrieved through a pipe that the driver sets while launching the command.

–passcmd cmd

The keystore password is retrieved by the running the external command defined in *cmd*. The command should print the password on its standard output without end of line. The password is retrieved through a pipe that the driver sets while launching the command.

–wallet-key-file file

Defines the path of a file which contains the wallet master key file.

COMMANDS

The start command

driver start [–management-port *PORT*] [–port *PORT*] [–connection *COUNT*][–upload *DIR*] [–tcp-no-delay] [–daemon] [–keystore *PATH*]

The `start` command allows to start the server and all applications that are registered to the web container. When a keystore is specified, it is first unlocked by using one of the unlock mechanism (password, GPG, …). Then, each application is configured by using its own configuration file and a subset of the keystore secured configuration. Once each application is configured, the web server container is started to listen to the TCP/IP port defined by the `--port`=*PORT* option. Applications are then started and they can serve HTTP requests through the web server container.

At the same time, a management port is created and used exclusively by the `stop` command to stop the web server container. The `start` command will wait on that management port for the `stop` command to be executed. The management port is configured by the `--management`=*PORT* option. The management port is local to the host and cannot be accessed remotely.

When the `--daemon` option is used, the server will first put itself in the background. This option is supported only under some Unix systems like *GNU/Linux* and *FreeBSD* and more generally every system where the `daemon(3)` C-library call is supported. On other systems, this option has no effect.

The `--tcp-no-delay` option is supported for recent version of Ada Web Server and configure the web server to use the `TCP_NO_DELAY` option of the TCP/IP stack (strongly recommended).

The `--upload`=*DIR* option indicates to the web server container the directory that can be used to store temporarily the uploaded files that the server receives.

The `--connection`=*COUNT* option controls the maximum number of active HTTP requests that the server can handle.

The setup command

driver setup [–management-port *PORT*] [–port *PORT*] [–connection *COUNT*][–upload *DIR*] [–tcp-no-delay] [–daemon] [–keystore *PATH*] *NAME*

The `setup` command is very close to the `start` command but it starts the Setup Application to configure the application by using a web browser.

The stop command

driver stop [–management-port *PORT*] [–keystore *PATH*]

The `stop` command is used to inform a running web server container to stop. The management port is used to connect to the web server container and stop it. The management port is local to the host and cannot be accessed remotely.

The management port is configured with the `--management-port`=*PORT* option.

The list command

driver list [–application *NAME*] [–keystore *PATH*] [–users] [–jobs] [–sessions] [–tables]

The `list` command is intended to help in looking at the application database and see some important information. Because the database is specific to each application, it may be necessary to indicate the application name by using the `--application`=*NAME* option.

The `--tables` option triggers the list of database tables with the number of entries they contain.

The `--users` option triggers the list of users that are registered in the application.

The `--sessions` option triggers the list of user connection sessions.

The `--jobs` option triggers the list of jobs that have been created and scheduled.

The info command

driver info [–application *NAME*] [–keystore *PATH*] [–long-lines]

The `info` command reports the current configuration used by the application. The configuration is extracted from the AWA default XML configuration files and can be overriden by the application specific configuration. The command allows to see what is the actual configuration. The configuration is printed with the configuration name and its associated value.

The list of configuration parameters are grouped in several categories:

- `Database configuration` gives the configuration properties for the database configuration.
- `Server faces configuration` lists the configuration use by the Ada Servlets and Ada Server Faces.
- `AWA Application` lists the core AWA configuration properties.

Depending on whether a module is used by the application, a number of modules are listed with their configuration.

The `--long-lines` option triggers the list of database tables with the number of entries they contain.

4.6.2 Integration

The `AWA.Commands` framework is split in several generic packages that must be instantiated. The `AWA.Commands.Drivers` generic package is the primary package that must be instantiated. It provides the core command line driver framework on top of which the commands are implemented. The instantiation needs two parameter: the name of the application and the type of the web server container. When using Ada Web Server, the following instantiation example can be used:

```
with Servlet.Server.Web;
with AWA.Commands.Drivers;
...
package Server_Commands is
   new AWA.Commands.Drivers
      (Driver_Name    => "atlas",
       Container_Type => Servlet.Server.Web.AWS_Container);
```

The `Driver_Name` is used to print the name of the command when some usage or help is printed. The `Container_Type` is used to declare the web container onto which applications are registered and which provides the web server core implementation. The web server container instance is available through the `WS` variable defined in the `Server_Commands` package.

Once the command driver is instantiated, it is necessary to instantiate each command that you wish to integrate in the final application. For example, to integrate the `start` command, you will do:

```
with AWA.Commands.Start;
...
package Start_Command is new AWA.Commands.Start (Server_Commands);
```

To integrate the `stop` command, you will do:

```
with AWA.Commands.Stop;
...
package Stop_Command is new AWA.Commands.Stop (Server_Commands);
```

The instantiation of one of the command, automatically registers the command to the command driver.

5 Users Module

The users module manages the creation, update, removal and authentication of users in an application. The module provides the foundations for user management in a web application.

A user can register himself by using a subscription form. In that case, a verification mail is sent and the user has to follow the verification link defined in the mail to finish the registration process. The user will authenticate using a password.

A user can also use an OAuth/OpenID account and be automatically authentified and registered to the application. By using an external authentication server, passwords are not stored in the application.

A user can have one or several permissions that allow to protect the application data. User permissions are managed by the Permissions.Module.

5.1 Integration

The User_Module manages the creation, update, removal of users in an application. It provides operations that are used by the user beans or other services to create and update wiki pages. An instance of the User_Module must be declared and registered in the AWA application. The module instance can be defined as follows:

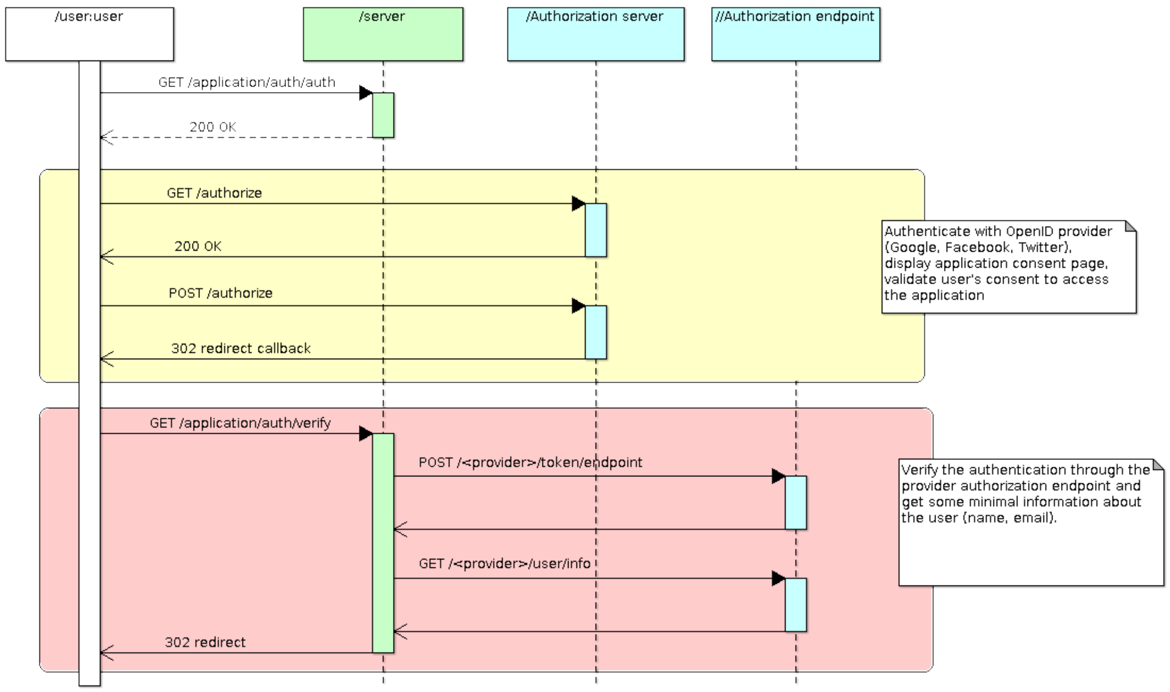
```
type Application is new AWA.Applications.Application with record
   User_Module : aliased AWA.Users.Modules.User_Module;
end record;
```

And registered in the Initialize_Modules procedure by using:

```
Register (App     => App.Self.all'Access,
          Name    => AWA.Users.Modules.NAME,
          Module => App.User_Module'Access);
```

5.2 OAuth Authentication Flow

The OAuth/OpenID authentication flow is implemented by using two servlets that participate in the authentication. A first servlet will start the OAuth/OpenID authentication by building the request that the user must use to authenticate through the OAuth/OpenID authorization server. This servlet is implemented by the AWA.Users.Servlets.Request_Auth_Servlet type. The servlet will respond to an HTTP GET request and it will redirect the user to the authorization server.



Authenticate with OpenID provider (Google, Facebook, Twitter), display application consent page, validate user's consent to access the application

Verify the authentication through the provider authorization endpoint and get some minimal information about the user (name, email).

Figure 13: *OAuth Authentication Flow*

The user will be authenticated by the OAuth/OpenID authorization server and when s/he grants the application to access his or her account, a redirection is made to the second servlet. The second servlet is implemented by `AWA.Users.Servlets.Verify_Auth_Servlet`. It is used to validate the authentication result by checking its validity with the OAuth/OpenID authorization endpoint. During this step, we can retrieve some minimal information that uniquely identifies the user such as a unique identifier that is specific to the OAuth/OpenID authorization server. It is also possible to retrieve the user's name and email address.

These two servlets are provided by the `User_Module` and they are registered under the `openid-auth` name for the first step and under the `openid-verify` name for the second step.

5.3 Configuration

The *users* module uses a set of configuration properties to configure the OpenID integration.

Name	Description
users.server_id	The server id when several instances are used.
	1

Name	Description
users.auth_key	An authentication key used to sign the authentication cookies.
	8ef60aad66977c68b12f4f8acab5a4e00a77f6e8
openid.realm	The REALM URL used by OpenID providers to verify the validity of the verification callback.
	#{app_url_base}/auth
openid.callback_url	The verification callback URI used by the OpenID provider to redirect the user after authentication.
	#{app_url_base}/auth/verify
openid.success_url	The URI where the user is redirected after a successful authentication.
	#{contextPath}/workspaces/main.html
auth.url.orange	Orange OpenID access point
	https://openid.orange.fr
auth.provider.orange	Auth provider to use for Orange
	openid
auth.url.yahoo	Yahoo! OpenID access point
	https://open.login.yahooapis.com/openid20/www.yahoo.com/xrds
auth.provider.yahoo	Auth provider to use for Yahoo!
	openid
auth.url.google	Google OpenID access point
	https://www.google.com/accounts/o8/id
auth.provider.google	Auth provider to use for Google
	openid
auth.url.facebook	Facebook OAuth access point
	https://www.facebook.com/dialog/oauth
auth.provider.facebook	Auth provider to use for Facebook
	facebook
facebook.callback_url	Facebook verify callback

Name	Description
	#{app_oauth_url_base}#{contextPath}/auth/verify
facebook.request_url	Facebook request OAuth token access point
	https://graph.facebook.com/oauth/access_token
facebook.scope	Facebook permission scope
	public_profile,email
facebook.client_id	Facebook API client ID
	#{app_facebook_client_id}
facebook.secret	Facebook API secret
	#{app_facebook_secret}
auth.url.google-plus	Google+ OAuth access point
	https://accounts.google.com/o/oauth2/auth
auth.provider.google-plus	Auth provider to use for Google+
	google-plus
google-plus.issuer	Google+ issuer identification
	accounts.google.com
google-plus.callback_url	Google+ verify callback
	#{app_oauth_url_base}#{contextPath}/auth/verify
google-plus.request_url	Google+ request OAuth token access point
	https://accounts.google.com/o/oauth2/token
google-plus.scope	Google+ permission scope
	openid profile email
google-plus.client_id	Google+ API client ID
	#{app_google_plus_client_id}
google-plus.secret	Google+ API secret
	#{app_google_plus_secret}
auth-filter.redirect	URI to redirect to the login page
	#{contextPath}/auth/login.html

Name	Description
verify-filter.redirect	URI to redirect to the login page
	#{contextPath}/auth/login.html

5.4 Ada Beans

Several bean types are provided to represent and manage the users. The user module registers the bean constructors when it is initialized. To use them, one must declare a bean definition in the application XML configuration.

Name	Description
login	This bean is used by the login form
register	This bean is used by the registration form
resetPassword	This bean is used by the reset password form
lostPassword	This bean is used by the lost password form
logout	This bean is used by the logout process
user	This bean allows to provide information about the current logged user.

5.5 Data model

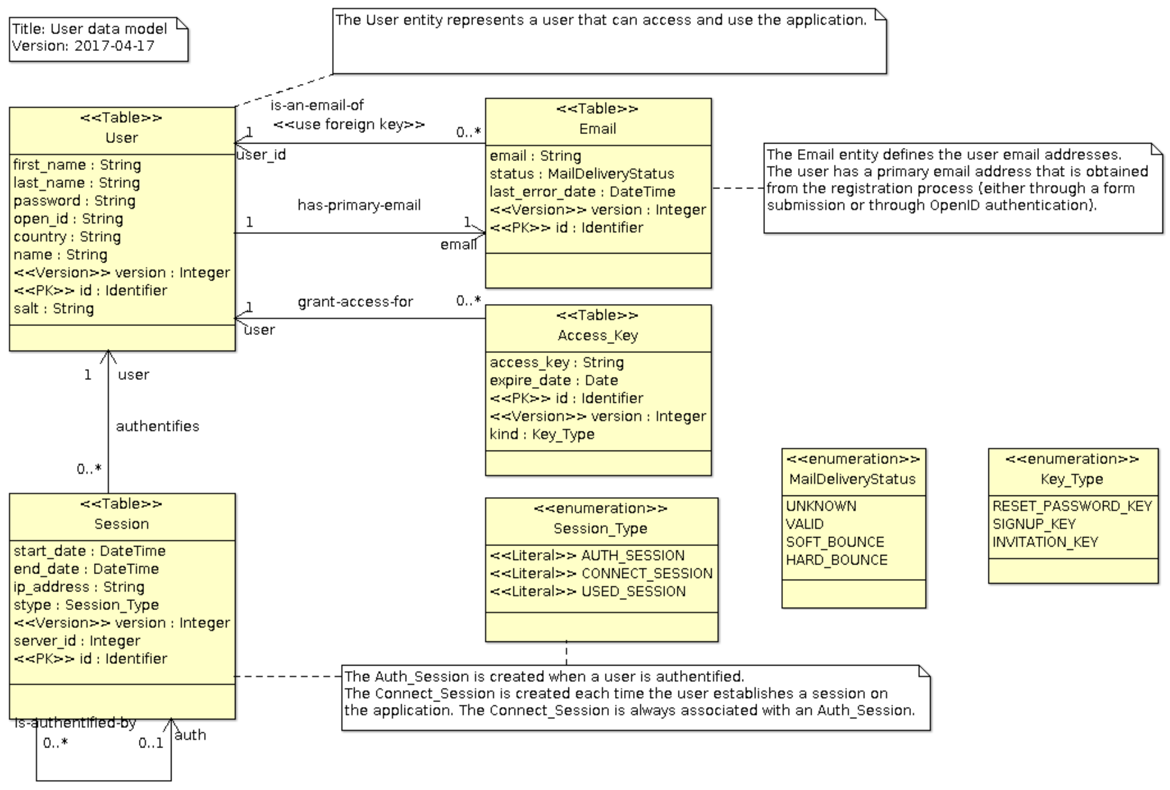

Title: User data model
Version: 2017-04-17
The User entity represents a user that can access and use the application.
<<Table>> User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
is-an-email-of
<<use foreign key>>
1
0..*
user_id
has-primary-email
1
1
email
grant-access-for
1
0..*
user
<<Table>> Email
email : String
status : MailDeliveryStatus
last_error_date : DateTime
<<Version>> version : Integer
<<PK>> id : Identifier
The Email entity defines the user email addresses.
The user has a primary email address that is obtained from the registration process (either through a form submission or through OpenID authentication).
<<Table>> Access_Key
access_key : String
expire_date : Date
<<PK>> id : Identifier
<<Version>> version : Integer
kind : Key_Type
1
user
authentifies
0..*
<<Table>> Session
start_date : DateTime
end_date : DateTime
ip_address : String
stype : Session_Type
<<Version>> version : Integer
server_id : Integer
<<PK>> id : Identifier
is-authentified-by
0..*
0..1
auth
<<enumeration>> Session_Type
<<Literal>> AUTH_SESSION
<<Literal>> CONNECT_SESSION
<<Literal>> USED_SESSION
<<enumeration>> MailDeliveryStatus
UNKNOWN
VALID
SOFT_BOUNCE
HARD_BOUNCE
<<enumeration>> Key_Type
RESET_PASSWORD_KEY
SIGNUP_KEY
INVITATION_KEY
The Auth_Session is created when a user is authentified.
The Connect_Session is created each time the user establishes a session on the application. The Connect_Session is always associated with an Auth_Session.

Figure 14

6 Jobs Module

The `jobs` module defines a batch job framework for modules to perform and execute long running and deferred actions. The `jobs` module is intended to help web application designers in implementing end to end asynchronous operation. A client schedules a job and does not block nor wait for the immediate completion. Instead, the client asks periodically or uses other mechanisms to check for the job completion.

6.1 Integration

To be able to use the `jobs` module, you will need to add the following line in your GNAT project file:

```
with "awa_jobs";
```

An instance of the `Job_Module` must be declared and registered in the AWA application. The module instance can be defined as follows:

```
with AWA.Jobs.Modules;
...
type Application is new AWA.Applications.Application with record
   Job_Module : aliased AWA.Jobs.Modules.Job_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App    => App.Self.all'Access,
          Name   => AWA.Jobs.Modules.NAME,
          Module => App.Job_Module'Access);
```

6.2 Writing a job

A new job type is created by implementing the `Execute` operation of the abstract `Job_Type` tagged record.

```
type Resize_Job is new AWA.Jobs.Job_Type with ...;
```

The `Execute` procedure must be implemented. It should use the `Get_Parameter` functions to retrieve the job parameters and perform the work. While the job is being executed, it can save result by using the `Set_Result` operations, save messages by using the `Set_Message` operations and report the progress by using `Set_Progress`. It may report the job status by using `Set_Status`.

```
procedure Execute (Job : in out Resize_Job) is
begin
    Job.Set_Result ("done", "ok");
end Execute;
```

6.3 Registering a job

The `jobs` module must be able to create the job instance when it is going to be executed. For this, a registration package must be instantiated:

```
package Resize_Def is new AWA.Jobs.Definition (Resize_Job);
```

and the job definition must be added:

```
AWA.Jobs.Modules.Register (Resize_Def.Create'Access);
```

6.4 Scheduling a job

To schedule a job, declare an instance of the job to execute and set the job specific parameters. The job parameters will be saved in the database. As soon as parameters are defined, call the `Schedule` procedure to schedule the job in the job queue and obtain a job identifier.

```
Resize : Resize_Job;
...
Resize.Set_Parameter ("file", "image.png");
Resize.Set_Parameter ("width", "32");
Resize.Set_Parameter ("height, "32");
Resize.Schedule;
```

6.5 Checking for job completion

After a job is scheduled, a unique identifier is allocated that allows to identify it. It is possible to query the status of the job by using the `Get_Job_Status` function:

```
Status : AWA.Jobs.Models.Job_Status_Type
  := AWA.Jobs.Services.Get_Job_Status (Resize.Get_Identifier);
```

6.6 Job Service

The `AWA.Jobs.Services` package defines the type abstractions and the core operation to define a job operation procedure, create and schedule a job and perform the job work when it is scheduled.

6.7 Ada Beans

Name	Description
jobHandler	The jobHandler is the bean that is created to execute a job.

6.8 Data Model

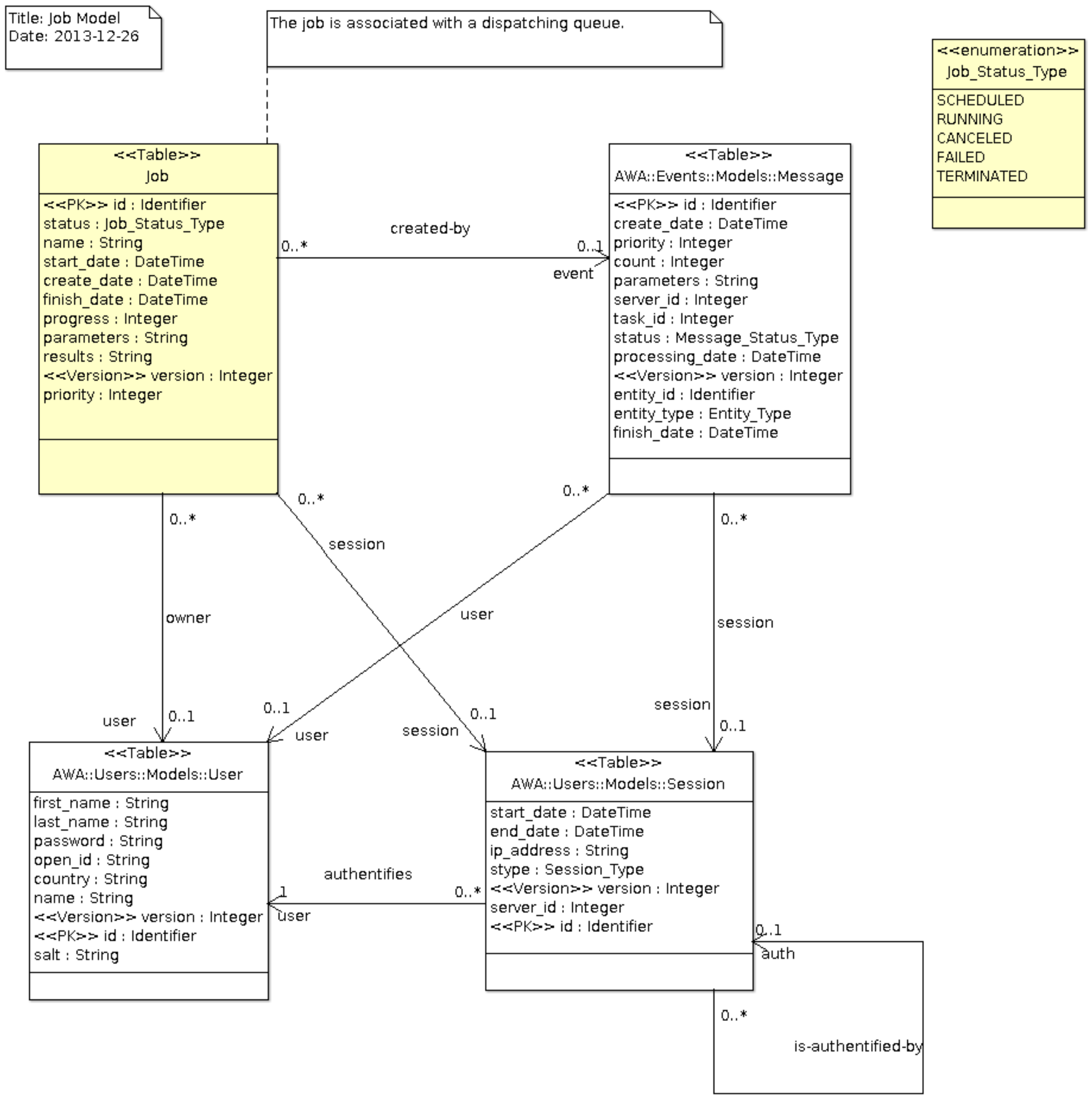

Title: Job Model
Date: 2013-12-26
The job is associated with a dispatching queue.
<<enumeration>>
Job_Status_Type
SCHEDULED
RUNNING
CANCELED
FAILED
TERMINATED
<<Table>>
Job
<<PK>> id : Identifier
status : Job_Status_Type
name : String
start_date : DateTime
create_date : DateTime
finish_date : DateTime
progress : Integer
parameters : String
results : String
<<Version>> version : Integer
priority : Integer
<<Table>>
AWA::Events::Models::Message
<<PK>> id : Identifier
create_date : DateTime
priority : Integer
count : Integer
parameters : String
server_id : Integer
task_id : Integer
status : Message_Status_Type
processing_date : DateTime
<<Version>> version : Integer
entity_id : Identifier
entity_type : Entity_Type
finish_date : DateTime
created-by
0..*
0..1
event
0..*
owner
user
0..1
0..*
session
0..1
session
0..*
user
0..1
user
0..*
session
session
0..1
<<Table>>
AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
<<Table>>
AWA::Users::Models::Session
start_date : DateTime
end_date : DateTime
ip_address : String
stype : Session_Type
<<Version>> version : Integer
server_id : Integer
<<PK>> id : Identifier
authentifies
1
user
0..*
0..1
auth
0..*
is-authentified-by

Figure 15

7 Mail Module

The `mail` module allows an application to format and send a mail to users. This module does not define any web interface. It provides a set of services and methods to send a mail when an event is received. All this is done through configuration. The module defines a set of specific ASF components to format and prepare the email.

7.1 Integration

To be able to use the `mail` module, you will need to add the following line in your GNAT project file:

```
with "awa_mail";
```

The `Mail_Module` type represents the mail module. An instance of the mail module must be declared and registered when the application is created and initialized. The module instance can be defined as follows:

```
type Application is new AWA.Applications.Application with record
   Mail_Module : aliased AWA.Mail.Modules.Mail_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App    => App.Self.all'Access,
          Name   => AWA.Mail.Modules.NAME,
          Module => App.Mail_Module'Access);
```

7.2 Configuration

The `mail` module needs some properties to configure the SMTP server.

Configuration	Default	Description
mail.smtp.host	localhost	Defines the SMTP server host name
mail.smtp.port	25	Defines the SMTP connection port
mail.smtp.enable	1	Defines whether sending email is enabled or not

7.3 Sending an email

Sending an email when an event is posted can be done by using an XML configuration. Basically, the `mail` module uses the event framework provided by AWA. The XML definition looks like:

```
<on-event name="user-register">
  <action>#{userMail.send}</action>
  <property name="template">/mail/register-user-message.xhtml</property
      >
</on-event>
```

With this definition, the mail template `/mail/register-user-message.xhtml` is formatted by using the event and application context when the `user-register` event is posted.

7.4 Components

The `AWA.Mail.Components` package defines several UI components that represent a mail message in an ASF view. The components allow the creation, formatting and sending of a mail message using the same mechanism as the application presentation layer. Example:

```
<f:view xmlns="mail:http://code.google.com/p/ada-awa/mail">
  <mail:message>
    <mail:subject>Welcome</mail:subject>
    <mail:to name="Iorek Byrnison">Iorek.Byrnison@svalbard.com</mail:to
        >
    <mail:body>
        ...
    </mail:body>
    <mail:attachment value="/images/mail-image.jpg"
       fileName="image.jpg"
       contentType="image/jpg"/>
  </mail:message>
</f:view>
```

When the view which contains these components is rendered, a mail message is built and initialized by rendering the inner components. The body and other components can use other application UI components to render useful content. The email is send after the `mail:message` has finished to render its inner children.

The `mail:subject` component describes the mail subject.

The `mail:to` component define the mail recipient. There can be several recepients.

The `mail:body` component contains the mail body.

The `mail:attachment` component allows to include some attachment.

7.4.1 Mail Recipients

The AWA.Mail.Components.Recipients package defines the UI components to represent the To, From, Cc and Bcc recipients.

The mail message is retrieved by looking at the parent UI component until a `UIMailMessage` component is found. The mail message recipients are initialized during the render response JSF phase, that is when Encode_End are called.

7.4.2 Mail Messages

The `AWA.Mail.Components.Messages` package defines the UI components to represent the email message with its recipients, subject and body.

The mail message is retrieved by looking at the parent UI component until a `UIMailMessage` component is found. The mail message recipients are initialized during the render response JSF phase, that is when `Encode_End` are called.

The `<mail:body>` component holds the message body. This component can include a facelet labeled `alternative` in which case it will be used to build the `text/plain` mail message. The default content type for `<mail:body>` is `text/html` but this can be changed by using the `type` attribute.

```
<mail:body type='text/html'>
   <facet name='alternative'>
      The text/plain mail message.
   </facet>
   The text/html mail message.
</mail:body>
```

7.4.3 Mail Attachments

The `AWA.Mail.Components.Attachments` package defines the UI components to represent a mail attachment. The mail attachment can be an external file or may be provided by an Ada bean object.

7.5 Ada Beans

Name	Description
userMail	Bean used to send an email with a specific template to the user.

8 Workspaces Module

The *workspaces* plugin defines a workspace area for other plugins. The workspace is intended to link together all the data objects that an application manages for a given user or group of users. A workspace is a possible mechanism to provide and implement multi-tenancy in a web application. By using the workspace plugin, application data from different customers can be kept separate from each other in the same database.

8.1 Events

The *workspaces* module provides several events that are posted when some action are performed.

8.1.1 invite-user

This event is posted when an invitation is created for a user. The event can be used to send the associated invitation email to the invitee. The event contains the following attributes:

key email name message inviter

8.1.2 accept-invitation

This event is posted when an invitation is accepted by a user.

8.2 Ada Beans

8.2.1 Beans

Name	Description
workspace	This bean allows to perform some general workspace actions
memberList	The list of workspace members.
inviteUser	The invitation bean.
workspaceMember	The workspace member bean.

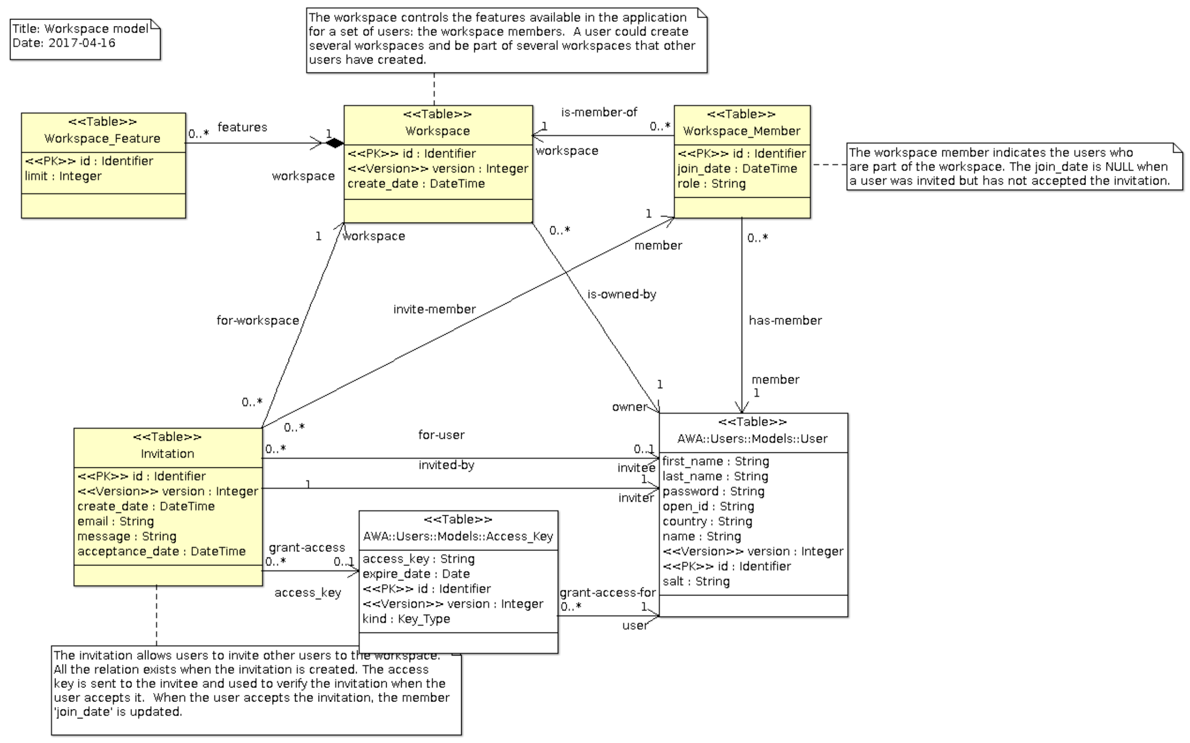
8.2.2 Permissions

Name	Entity type	Description
workspace-create	awa_workspace	Permission to create a workspace.
workspace-invite-user	awa_workspace	Permission to invite a user in the workspace.
workspace-delete-user	awa_workspace	Permission to delete a user from the workspace.
workspaces-create	awa_workspace	

8.2.3 Configuration

Name	Description
workspaces.permissions_list	
	blog-create,wiki-space-create
workspaces.allow_workspace_create	
	0

8.3 Data Model


Title: Workspace model
Date: 2017-04-16

The workspace controls the features available in the application for a set of users: the workspace members. A user could create several workspaces and be part of several workspaces that other users have created.

The workspace member indicates the users who are part of the workspace. The join_date is NULL when a user was invited but has not accepted the invitation.

The invitation allows users to invite other users to the workspace. All the relation exists when the invitation is created. The access key is sent to the invitee and used to verify the invitation when the user accepts it. When the user accepts the invitation, the member 'join_date' is updated.

Figure 16

9 Storages Module

The `storages` module provides a set of storage services allowing an application to store data files, documents, images in a persistent area. The persistent store can be on a file system, in the database or provided by a remote service such as Amazon Simple Storage Service.

9.1 Integration

To be able to use the `storages` module, you will need to add the following line in your GNAT project file:

```
with "awa_storages";
```

The `Storage_Module` type represents the storage module. An instance of the storage module must be declared and registered when the application is created and initialized. The storage module is associated with the storage service which provides and implements the storage management operations. An instance of the `Storage_Module` must be declared and registered in the AWA application. The module instance can be defined as follows:

```
with AWA.Storages.Modules;
...
type Application is new AWA.Applications.Application with record
   Storage_Module : aliased AWA.Storages.Modules.Storage_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App     => App.Self.all'Access,
          Name    => AWA.Storages.Modules.NAME,
          Module => App.Storage_Module'Access);
```

9.2 Permissions

Name	Entity type	Description
folder-create	awa_workspace	
storage-create	awa_workspace	
storage-delete	awa_workspace	

9.3 Configuration

The storages module defines the following configuration parameters:

Name	Description
storages.storage_root	The path of the directory that contains storage files stored on the local filesystem.
	storage
storages.tmp_storage_root	The path of the directory that contains temporary storage files on the local filesystem.
	tmp
storages.database_max_size	The maximum size of documents store in the database storage.
	100000

9.4 Creating a storage

A data in the storage is represented by a Storage_Ref instance. The data itself can be physically stored in a file system (FILE mode), in the database (DATABASE mode) or on a remote server (URL mode). To put a file in the storage space, first create the storage object instance:

```ada
Data : AWA.Storages.Models.Storage_Ref;
```

Then setup the storage mode that you want. The storage service uses this information to save the data in a file, in the database or in a remote service (in the future). To save a file in the store, we can use the Save operation of the storage service. It will read the file and put in in the corresponding persistent store (the database in this example).

```ada
Service.Save (Into => Data, Path => Path_To_The_File,
              Storage => AWA.Storages.Models.DATABASE);
```

Upon successful completion, the storage instance Data will be allocated a unique identifier that can be retrieved by Get_Id or Get_Key.

9.5 Getting the data

Several operations are defined to retrieve the data. Each of them has been designed to optimize the retrieval and

- The data can be retrieved in a local file. This mode is useful if an external program must be launched and be able to read the file. If the storage mode of the data is `FILE`, the path of the file on the storage file system is used. For other storage modes, the file is saved in a temporary file. In that case the `Store_Local` database table is used to track such locally saved data.
- The data can be returned as a stream. When the application has to read the data, opening a read stream connection is the most efficient mechanism.

9.6 Local file

To access the data by using a local file, we must define a local storage reference:

```
Data : AWA.Storages.Models.Store_Local_Ref;
```

and use the `Load` operation with the storage identifier. When loading locally we also indicate whether the file will be read or written. A file that is in `READ` mode can be shared by several tasks or processes. A file that is in `WRITE` mode will have a specific copy for the caller. An optional expiration parameter indicate when the local file representation can expire.

```
Service.Load (From => Id, Into => Data, Mode => READ, Expire => ONE_DAY
    );
```

Once the load operation succeeded, the data is stored on the file system and the local path is obtained by using the `Get_Path` operation:

```
Path : constant String := Data.Get_Path;
```

9.7 Storage Service

The Storage_Service provides the operations to access and use the persisent storage. It controls the permissions that grant access to the service for users.

Other modules can be notified of storage changes by registering a listener on the storage module.

9.8 Store Service

The `AWA.Storages.Stores` package defines the interface that a store must implement to be able to save and retrieve a data content. The store can be a file system, a database or a remote store service.

9.8.1 Database store

The `AWA.Storages.Stores.Databases` store uses the database to save a data content. The data is saved in a specific table in a database blob column. The database store uses another store service to temporarily save the data content in a local file when the application needs a file access to the data.

9.8.2 File System store

The `AWA.Storages.Stores.Files` store uses the file system to save a data content. Files are stored in a directory tree whose path is created from the workspace identifier and the storage identifier. The layout is such that files belonged to a given workspace are stored in the same directory sub-tree.

The root directory of the file system store is configured through the storage_root and tmp_storage_root configuration properties.

9.9 Ada Beans

Name	Description
storageFolder	This bean allows to create a storage folder.
uploadFile	This bean allows to upload a new file in the storage space.
folderList	This bean gives the list of storage folders in the workspace.
storageList	This bean gives the list of storage files associated with a given folder.
storageInfo	This bean gives some information about a document and its folder.

AWA.Storages.Models.Storage_Info

The list of documents for a given folder.

Type	Ada	Name	Description
	Identifier	id	the storage identifier.

Type	Ada	Name	Description
	String	name	the file name.
	Date	create_date	the file creation date.
	String	uri	the file storage URI.
	AWA.Storages.Models.Storage_Type	storage	the file storage URI.
	String	mime_type	the file mime type.
	Integer	file_size	the file size.
	Boolean	is_public	whether the document is public or not.
	String	user_name	the user name who uploaded the document.
	Integer	thumb_width	the image thumbnail width (or 0).
	Integer	thumb_height	the image thumbnail height (or 0).
	Identifier	thumbnail_id	the image thumbnail identifier.

AWA.Storages.Models.Folder_Info

The list of folders.

Type	Ada	Name	Description
	Identifier	id	the folder identifier.
	String	name	the folder name.
	Date	create_date	the blog creation date.

9.10 Storage Servlet

The Storage_Servlet type is the servlet that allows to retrieve the file content that was uploaded.

9.11 Queries

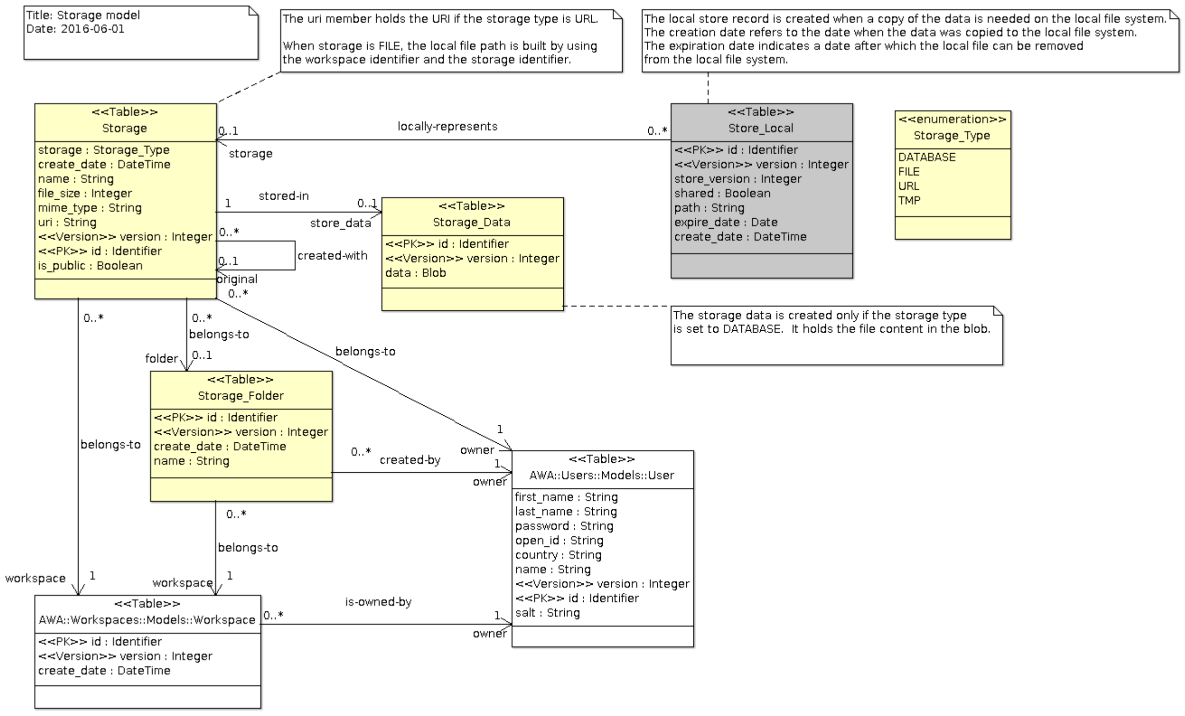
Name	Description
storage-list	Get a list of storage files for a given folder.

Name	Description
storage-folder-list	Get a list of storage folders that a user can see.

Name	Description
storage-get-data	Get the data content of the storage object.
storage-get-local	Get the local data storage that can be used to read locally a storage object.
storage-get-storage	Get the local data storage that can be used to read locally a storage object.
storage-delete-local	Delete the local storage data

Name	Description
storage-info	Get the description of a document.

9.12 Data model


Title: Storage model
Date: 2016-06-01

The uri member holds the URI if the storage type is URL.

When storage is FILE, the local file path is built by using the workspace identifier and the storage identifier.

The local store record is created when a copy of the data is needed on the local file system. The creation date refers to the date when the data was copied to the local file system. The expiration date indicates a date after which the local file can be removed from the local file system.

The storage data is created only if the storage type is set to DATABASE. It holds the file content in the blob.

Figure 17

10 Images Module

The `images` module is an extension of the Storages Module that identifies images and provides thumbnails as well as resizing of the original image.

The `images` module uses several other modules:

- the Storage Module to store and manage image content,
- the Jobs Module to schedule image thumbnail generation.

10.1 Integration

To be able to use the `Images` module, you will need to add the following line in your GNAT project file:

```
with "awa_images";
```

The `Image_Module` type represents the image module. An instance of the image module must be declared and registered when the application is created and initialized. The image module is associated with the image service which provides and implements the image management operations.

```
with AWA.Images.Modules;
...
type Application is new AWA.Applications.Application with record
   Image_Module : aliased AWA.Images.Modules.Image_Module;
end record;
```

And it is registered in the `Initialize_Modules` procedure by using:

```
Register (App    => App.Self.all'Access,
          Name   => AWA.Images.Modules.NAME,
          Module => App.Image_Module'Access);
```

When the image module is initialized, it registers itself as a listener to the storage module to be notified when a storage file is created, updated or removed. When a file is added, it looks at the file type and extracts the image information if the storage file is an image.

10.2 Configuration

The `Images` module defines the following configuration parameters:

Name	Description
images.thumbnail_command	The command to execute to generate an image thumbnail for the Images module.
	convert -verbose -resize #{width}x#{height} -background white -gravity center -extent #{width}x#{height} -format jpg -quality 75 #{src} #{dst}

10.3 Ada Beans

The `Image_List_Bean` type is used to represent a list of image stored in a folder.

The `Image_Bean` type holds all the data to give information about an image.

Name	Description
storageFolder	This bean allows to create a storage folder.
imageList	This bean gives the list of images associated with a given folder.

Name	Description
imageInfo	This bean gives the information about an image.

AWA.Images.Models.Image_Bean

The information about an image.

Type	Ada	Name	Description
	Identifier	folder_id	the image folder identifier.
	String	folder_name	the image folder name.
	Identifier	id	the image file identifier.
	String	name	the image file name.
	Date	create_date	the file creation date.
	String	uri	the file storage URI.
	AWA.Storages.Models.Storage_Type	storage	the file storage URI.
	String	mime_type	the file mime type.
	Integer	file_size	the file size.
	Boolean	is_public	whether the image is public.
	Integer	width	the image width.
	Integer	height	the image height.

AWA.Images.Models.Image_Info

The list of images for a given folder.

Type	Ada	Name	Description
	Identifier	id	the storage identifier which contains the image data.
	String	name	the image file name.
	Date	create_date	the image file creation date.
	String	uri	the image file storage URI.

Type	Ada	Name	Description
	Integer	storage	the image file storage URI.
	String	mime_type	the image file mime type.
	Integer	file_size	the image file size.
	Integer	width	the image width.
	Integer	height	the image height.
	Integer	thumb_width	the image thumbnail width.
	Integer	thumb_height	the image thumbnail height.
	Identifier	thumbnail_id	the image thumbnail identifier.

10.4 Queries

Name	Description
image-info	Get the description of an image.

Name	Description
image-list	Get a list of images for a given folder.

10.5 Data model

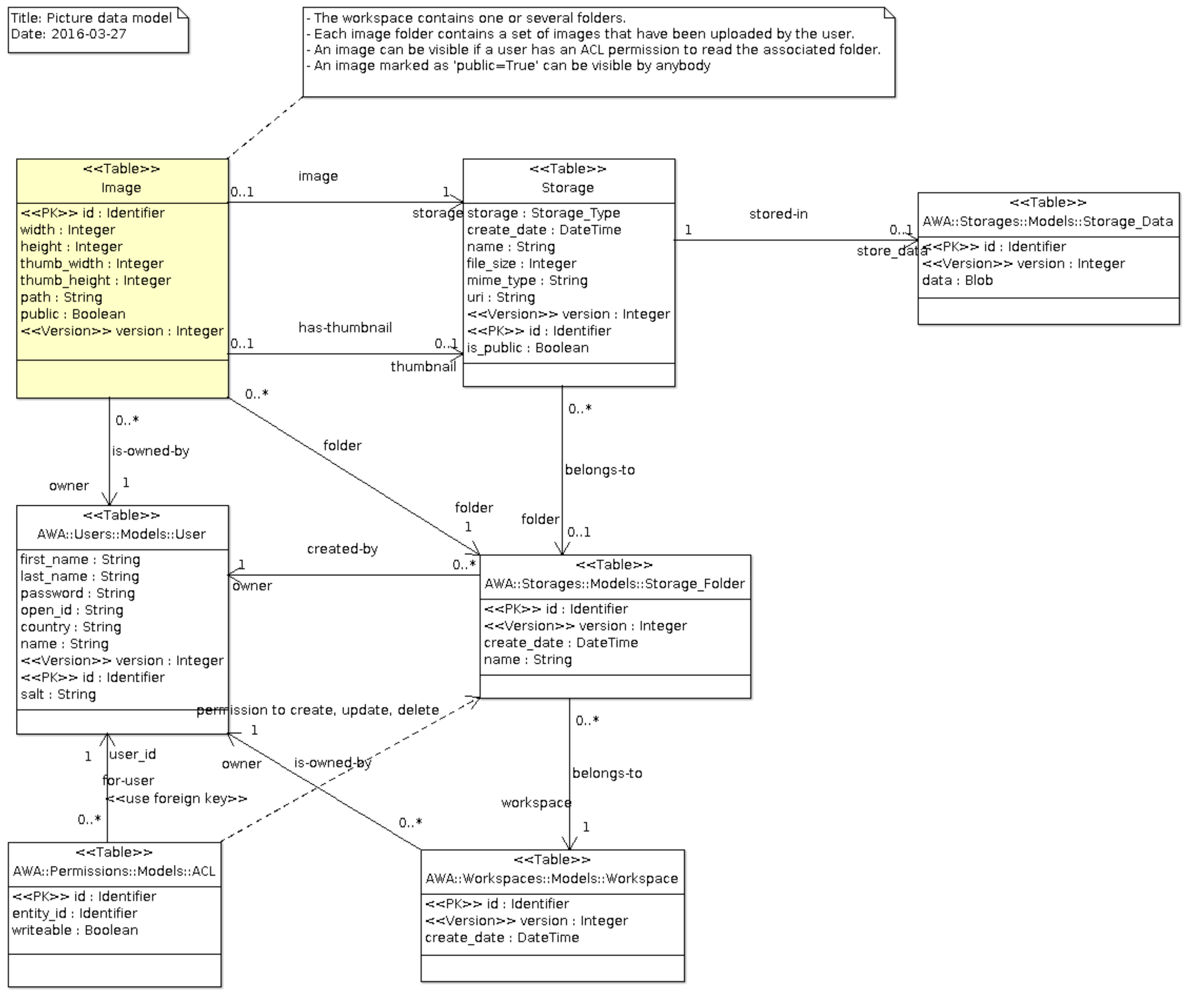

Title: Picture data model
Date: 2016-03-27
- The workspace contains one or several folders.
- Each image folder contains a set of images that have been uploaded by the user.
- An image can be visible if a user has an ACL permission to read the associated folder.
- An image marked as 'public=True' can be visible by anybody
<<Table>> Image
<<PK>> id : Identifier
width : Integer
height : Integer
thumb_width : Integer
thumb_height : Integer
path : String
public : Boolean
<<Version>> version : Integer
<<Table>> Storage
storage : Storage_Type
create_date : DateTime
name : String
file_size : Integer
mime_type : String
uri : String
<<Version>> version : Integer
<<PK>> id : Identifier
is_public : Boolean
<<Table>> AWA::Storages::Models::Storage_Data
<<PK>> id : Identifier
<<Version>> version : Integer
data : Blob
<<Table>> AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
<<Table>> AWA::Storages::Models::Storage_Folder
<<PK>> id : Identifier
<<Version>> version : Integer
create_date : DateTime
name : String
<<Table>> AWA::Permissions::Models::ACL
<<PK>> id : Identifier
entity_id : Identifier
writeable : Boolean
<<Table>> AWA::Workspaces::Models::Workspace
<<PK>> id : Identifier
<<Version>> version : Integer
create_date : DateTime
image
storage
has-thumbnail
thumbnail
stored-in
store_data
is-owned-by
owner
folder
belongs-to
created-by
permission to create, update, delete
user_id
for-user
<<use foreign key>>
workspace

Figure 18

11 Wikis Module

The `Wikis` module provides a complete wiki system which allows users to create their own wiki environment with their wiki pages.

11.1 Integration

To be able to use the `Wikis` module, you will need to add the following line in your GNAT project file:

```
with "awa_wikis";
```

The `Wiki_Module` manages the creation, update, removal of wiki pages in an application. It provides operations that are used by the wiki beans or other services to create and update wiki pages. An instance of the `Wiki_Module` must be declared and registered in the AWA application. The module instance can be defined as follows:

```
with AWA.Wikis.Modules;
...
type Application is new AWA.Applications.Application with record
   Wiki_Module : aliased AWA.Wikis.Modules.Wiki_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App    => App.Self.all'Access,
          Name   => AWA.Wikis.Modules.NAME,
          URI    => "wikis",
          Module => App.Wiki_Module'Access);
```

11.2 Configuration

Name	Description
wikis.image_prefix	The URL base prefix to be used for Wiki images.
	#{contextPath}/wikis/images/
wikis.page_prefix	The URL base prefix to be used for Wiki pages.
	#{contextPath}/wikis/view/
wikis.wiki_copy_list	A list of wiki page ID top copy when a new wiki space is created.

Name	Description

11.3 Events

The `wikis` exposes a number of events which are posted when some action are performed at the service level.

Event name	Description
wiki-create-page	This event is posted when a new wiki page is created.
wiki-create-content	This event is posted when a new wiki page content is created.
	Each time a wiki page is modified, a new wiki page content
	is created and this event is posted.

11.4 Ada Beans

Several bean types are provided to represent and manage the blogs and their posts. The blog module registers the bean constructors when it is initialized. To use them, one must declare a bean definition in the application XML configuration.

Name	Description
wikiView	The wiki page with all its information to display it.
wikiImageInfo	The information about an image used by a wiki page.
wikiPageInfo	The wiki page information bean gives the various statistics and information about a wiki page.
wikiFormatList	A localized list of wiki page formats to be used for a f:selectItems
adminWiki	The list of wikis and pages that the current user can access and update.
adminWikiSpace	The wiki space bean to create and edit the wiki space configuration.
wikiPage	The wiki page bean gives the full content and information about a wiki page.
wikiList	The list of wiki pages.
wikiVersionList	The list of wiki page versions.

Name	Description
wikiTagSearch	The wiki tag search bean.
wikiTagCloud	The list of tags associated with a wiki page entities.
wikiTags	The wiki tag editor bean.
wikiPageStats	The counter statistics for a wiki page

AWA.Wikis.Models.Wiki_View_Info

The information about a wiki page.

Type	Ada	Name	Description
	Identifier	id	the wiki page identifier.
	String	name	the wiki page name.
	String	title	the wiki page title.
	Boolean	is_public	whether the wiki is public.
	Nullable_Integer	version	the last version.
	Nullable_Integer	read_count	the number of times the page was displayed.
	Nullable_Date	date	the wiki page creation date.
	AWA.Wikis.Models.Format_Type	format	the wiki page format.
	String	content	the wiki page content.
	String	save_comment	the wiki version comment.
	String	left_side	the wiki page left side panel.
	String	right_side	the wiki page right side panel.
	AWA.Wikis.Models.Format_Type	side_format	the wiki side format.
	String	author	the wiki page author.
	Identifier	acl_id	the acl Id if there is one.

AWA.Wikis.Models.Wiki_Page_Info

The information about a wiki page.

Type	Ada	Name	Description
	Identifier	id	the wiki page identifier.
	String	name	the wiki page name.
	String	title	the wiki page title.
	Boolean	is_public	whether the wiki is public.
	Integer	last_version	the last version.
	Integer	read_count	the read count.
	Date	create_date	the wiki creation date.
	String	author	the wiki page author.

AWA.Wikis.Models.Wiki_Version_Info

The information about a wiki page version.

Type	Ada	Name	Description
	Identifier	id	the wiki page identifier.
	String	comment	the wiki page version comment.
	Date	create_date	the wiki page creation date.
	Integer	page_version	the page version.
	String	author	the wiki page author.

AWA.Wikis.Models.Wiki_Info

The list of wikis.

Type	Ada	Name	Description
	Identifier	id	the wiki space identifier.
	String	name	the wiki name.
	Boolean	is_public	whether the wiki is public.
	Date	create_date	the wiki creation date.
	Integer	page_count	the number of pages in the wiki.

Type	Ada	Name	Description

11.5 Queries

Name	Description
wiki-page	Get the content of a wiki page.
wiki-page-id	Get the content of a wiki page.
wiki-page-content	Get only the content of a wiki page (for template evaluation).
wiki-page-name-count	Count the occurence of a wiki page name

Name	Description
wiki-page-list	Get the list of wiki pages
wiki-page-tag-list	Get the list of wiki pages filtered by a tag

Name	Description
wiki-version-list	Get the list of wiki page versions

Name	Description
wiki-list	Get the list of wikis that the current user can update

Name	Description
wiki-image-get-data	Get the data content of the Wiki image (original image).

Name	Description
wiki-image-width-get-data	Get the data content of the Wiki image for an image with a given width.
wiki-image-height-get-data	Get the data content of the Wiki image for an image with a given height.

Name	Description
wiki-image	Get the description of an image used in a wiki page.

Name	Description
page-access-stats	Get statistics about the wiki page access.

11.6 Data model

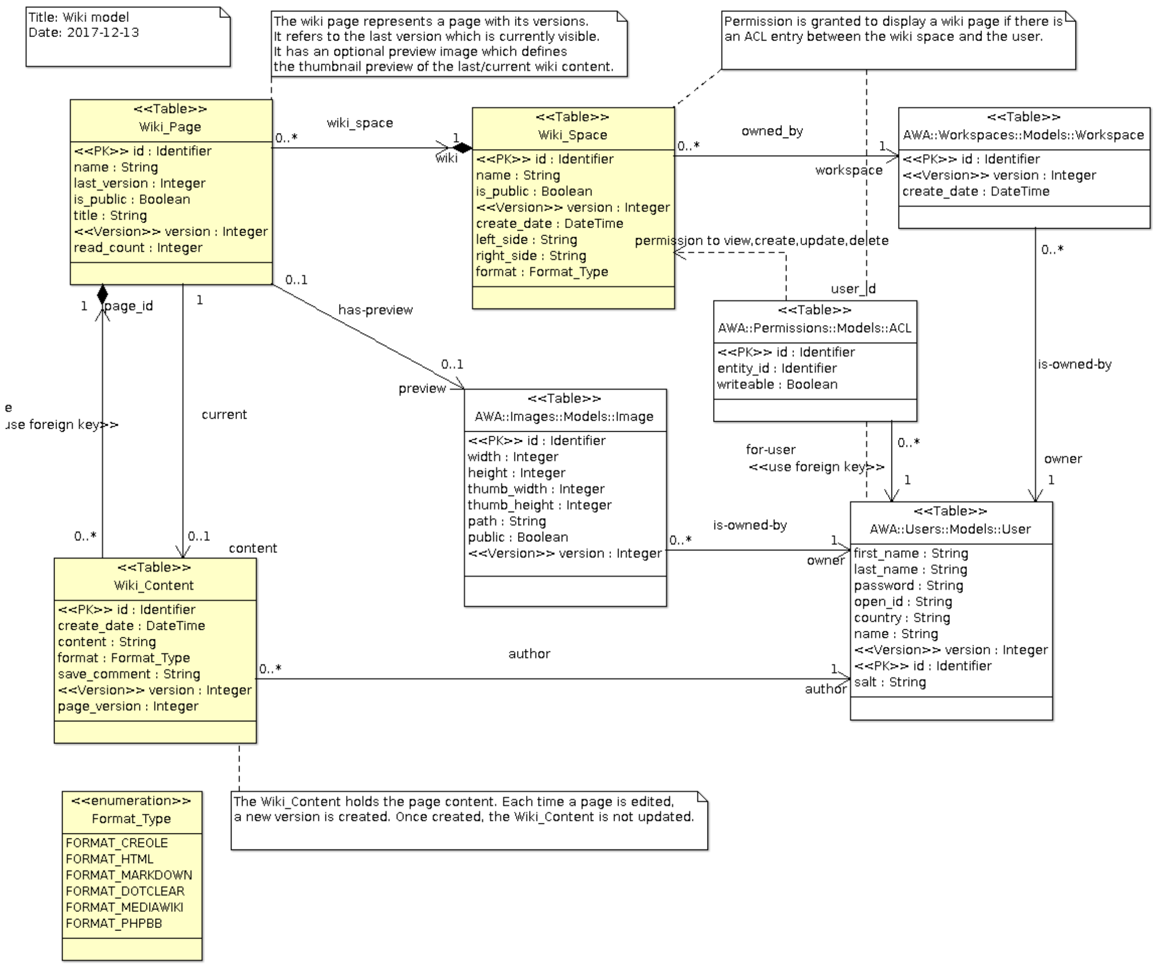

Title: Wiki model
Date: 2017-12-13
The wiki page represents a page with its versions.
It refers to the last version which is currently visible.
It has an optional preview image which defines
the thumbnail preview of the last/current wiki content.
Permission is granted to display a wiki page if there is
an ACL entry between the wiki space and the user.
<<Table>>
Wiki_Page
<<PK>> id : Identifier
name : String
last_version : Integer
is_public : Boolean
title : String
<<Version>> version : Integer
read_count : Integer
wiki_space
0..*
1
wiki
<<Table>>
Wiki_Space
<<PK>> id : Identifier
name : String
is_public : Boolean
<<Version>> version : Integer
create_date : DateTime
left_side : String
right_side : String
format : Format_Type
owned_by
0..*
1
workspace
<<Table>>
AWA::Workspaces::Models::Workspace
<<PK>> id : Identifier
<<Version>> version : Integer
create_date : DateTime
permission to view,create,update,delete
0..*
user_id
<<Table>>
AWA::Permissions::Models::ACL
<<PK>> id : Identifier
entity_id : Identifier
writeable : Boolean
is-owned-by
0..1
has-preview
0..1
preview
1
page_id
1
current
<<use foreign key>>
<<Table>>
AWA::Images::Models::Image
<<PK>> id : Identifier
width : Integer
height : Integer
thumb_width : Integer
thumb_height : Integer
path : String
public : Boolean
<<Version>> version : Integer
for-user
<<use foreign key>>
0..*
1
owner
1
is-owned-by
0..*
1
owner
<<Table>>
AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
0..*
0..1
content
<<Table>>
Wiki_Content
<<PK>> id : Identifier
create_date : DateTime
content : String
format : Format_Type
save_comment : String
<<Version>> version : Integer
page_version : Integer
author
0..*
1
author
<<enumeration>>
Format_Type
FORMAT_CREOLE
FORMAT_HTML
FORMAT_MARKDOWN
FORMAT_DOTCLEAR
FORMAT_MEDIAWIKI
FORMAT_PHPBB
The Wiki_Content holds the page content. Each time a page is edited,
a new version is created. Once created, the Wiki_Content is not updated.

Figure 19

12 Blogs Module

The `blogs` module is a small blog application which allows users to publish articles. A user may own several blogs, each blog having a name and its own base URI. Within a blog, the user may write articles and publish them. Once published, the articles are visible to anonymous users.

The `blogs` module uses several other modules:

- the Counters Module to track page display counter to a blog post,
- the Tags Module to associate one or several tags to a blog post,
- the Comments Module to allow users to write comments on a blog post,
- the Images Module to easily add images in blog post.

12.1 Integration

To be able to use the `Blogs` module, you will need to add the following line in your GNAT project file:

```
with "awa_blogs";
```

The `Blog_Module` type manages the creation, update, removal of blog posts in an application. It provides operations that are used by the blog beans or other services to create and update posts. An instance of the `Blog_Module` must be declared and registered in the AWA application. The module instance can be defined as follows:

```
with AWA.Blogs.Modules;
...
type Application is new AWA.Applications.Application with record
   Blog_Module : aliased AWA.Blogs.Modules.Blog_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App    => App.Self.all'Access,
          Name   => AWA.Blogs.Modules.NAME,
          URI    => "blogs",
          Module => App.Blog_Module'Access);
```

12.2 Ada Beans

Several bean types are provided to represent and manage the blogs and their posts. The blog module registers the bean constructors when it is initialized. To use them, one must declare a bean definition in the application XML configuration.

Name	Description
post	This bean describes a blog post for the creation or the update
postList	This bean describes a blog post for the creation or the update
postStatusList	A localized list of post statuses to be used for a f:selectItems
postAccessStats	The counter statistics for a blog post
blogFormatList	A localized list of blog post formats to be used for a f:selectItems
adminBlog	The list of blogs and posts that the current user can access and update.
blog	Information about the current blog.
feed_blog	Information about the RSS feed blog.
blogTagSearch	The blog tag search bean.
blogTagCloud	A list of tags associated with all post entities.
postComments	A list of comments associated with a post.
postAdminComme	A list of all comments associated with a post (for admin purposes).
postNewComment	The bean to allow a user to post a new comment. This is a specific bean because the permission to create a new post is different from other permissions. If the permission is granted, the comment will be created and put in the COMMENT_WAITING state. An email will be sent to the post author for approval.
blogPublishComm	The bean to allow the blog administrator to change the publication status of a comment.
blogDeleteComment	The bean to allow the blog administrator to change the publication status of a comment.
commentEdit	The bean to allow a user to edit the comment. This is a specific bean because the permission to edit the comment is different from other permissions.
blogStats	This bean provides statistics about the blog

AWA.Blogs.Models.Admin_Post_Info

The Admin_Post_Info describes a post in the administration interface.

Type	Ada	Name	Description
	Identifier	id	the post identifier.
	String	title	the post title.
	String	uri	the post uri.
	Date	date	the post publish date.
	AWA.Blogs.Models.Post_Status_Type	status	the post status.
	Natural	read_count	the number of times the post was read.
	String	username	the user name.
	Natural	comment_count	the number of comments for this post.

AWA.Blogs.Models.Post_Info

The Post_Info describes a post to be displayed in the blog page

Type	Ada	Name	Description
	Identifier	id	the post identifier.
	String	title	the post title.
	String	uri	the post uri.
	Date	date	the post publish date.
	String	username	the user name.
	AWA.Blogs.Models.Format_Type	format	the post page format.
	String	summary	the post summary.
	String	text	the post text.
	Boolean	allow_comments	the post allows to add comments.
	Natural	comment_count	the number of comments for this post.

AWA.Blogs.Models.Comment_Info

The comment information.

Type	Ada	Name	Description
	Identifier	id	the comment identifier.
	Identifier	post_id	the post identifier.
	String	title	the post title.
	String	author	the comment author's name.
	String	email	the comment author's email.
	Date	date	the comment date.
	AWA.Comments.Models.Status_Type	status	the comment status.

AWA.Blogs.Models.Blog_Info

The list of blogs.

Type	Ada	Name	Description
	Identifier	id	the blog identifier.
	String	title	the blog title.
	String	uid	the blog uuid.
	Date	create_date	the blog creation date.
	Integer	post_count	the number of posts published.

12.3 Queries

Name	Description
blog-admin-post-list	Get the list of blog posts
blog-admin-post-list-date	Get the list of blog posts

Name	Description
blog-post-list	Get the list of public visible posts
blog-post-tag-list	Get the list of public visible posts filtered by a tag

Name	Description
comment-list	Get the list of comments associated with given database entity

Name	Description
blog-list	Get the list of blogs that the current user can update

Name	Description
blog-tag-cloud	Get the list of tags associated with all the database entities of a given type

Name	Description
blog-image-get-data	Get the data content of the Wiki image (original image).
blog-image-width-get-data	Get the data content of the Wiki image for an image with a given width.
blog-image-height-get-data	Get the data content of the Wiki image for an image with a given height.

Name	Description
blog-image	Get the description of an image used in a blog post.

Name	Description
post-publish-stats	Get statistics about the post publication on a blog.
post-access-stats	Get statistics about the post publication on a blog.

12.4 Data model

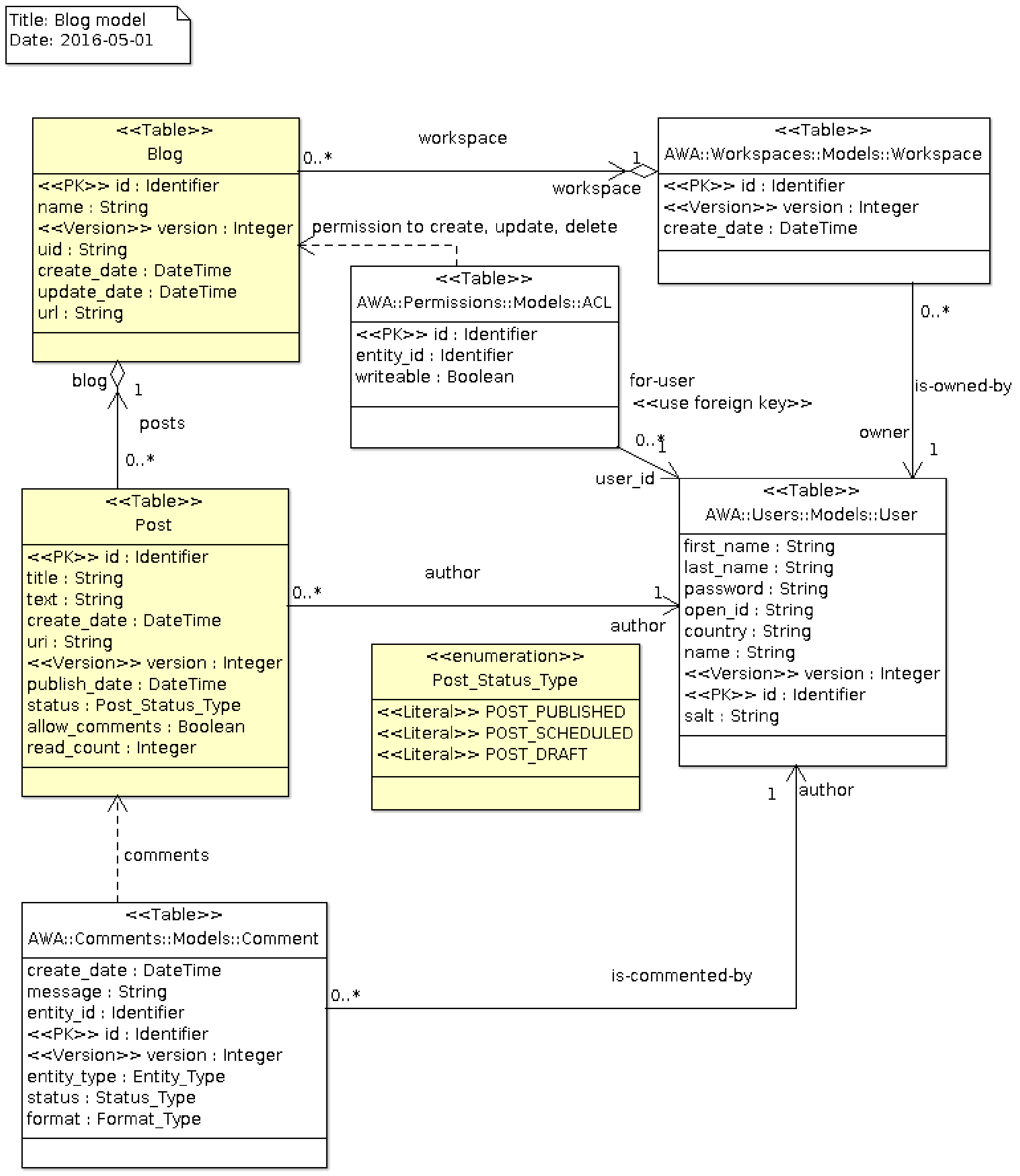

Title: Blog model
Date: 2016-05-01
<<Table>>
Blog
<<PK>> id : Identifier
name : String
<<Version>> version : Integer
uid : String
create_date : DateTime
update_date : DateTime
url : String
workspace
0..*
1
workspace
<<Table>>
AWA::Workspaces::Models::Workspace
<<PK>> id : Identifier
<<Version>> version : Integer
create_date : DateTime
permission to create, update, delete
<<Table>>
AWA::Permissions::Models::ACL
<<PK>> id : Identifier
entity_id : Identifier
writeable : Boolean
for-user
<<use foreign key>>
0..*
1
user_id
0..*
is-owned-by
owner
1
blog
1
posts
0..*
<<Table>>
Post
<<PK>> id : Identifier
title : String
text : String
create_date : DateTime
uri : String
<<Version>> version : Integer
publish_date : DateTime
status : Post_Status_Type
allow_comments : Boolean
read_count : Integer
author
0..*
1
author
<<Table>>
AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
<<enumeration>>
Post_Status_Type
<<Literal>> POST_PUBLISHED
<<Literal>> POST_SCHEDULED
<<Literal>> POST_DRAFT
1
author
comments
<<Table>>
AWA::Comments::Models::Comment
create_date : DateTime
message : String
entity_id : Identifier
<<PK>> id : Identifier
<<Version>> version : Integer
entity_type : Entity_Type
status : Status_Type
format : Format_Type
0..*
is-commented-by

Figure 20

13 Counters Module

The `counters` module defines a general purpose counter service that allows to associate counters to database entities. For example it can be used to track the number of times a blog post or a wiki page is accessed. The `counters` module maintains the counters in a table on a per-day and per-entity basis. It allows to update the full counter in the target database entity table.

13.1 Integration

The `Counter_Module` manages the counters associated with database entities. To avoid having to update the database each time a counter is incremented, counters are kept temporarily in a `Counter_Table` protected type. The table contains only the partial increments and not the real counter values. Counters are flushed when the table reaches some limit, or, when the table is oldest than some limit. Counters are associated with a day so that it becomes possible to gather per-day counters. The table is also flushed when a counter is incremented in a different day.

To be able to use the `Counters` module, you will need to add the following line in your GNAT project file:

```
with "awa_counters";
```

An instance of the `Counter_Module` must be declared and registered in the AWA application. The module instance can be defined as follows:

```
with AWA.Counters.Modules;
...
type Application is new AWA.Applications.Application with record
   Counter_Module : aliased AWA.Counters.Modules.Counter_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App     => App.Self.all'Access,
          Name    => AWA.Counters.Modules.NAME,
          Module  => App.Counter_Module'Access);
```

13.2 Configuration

The `counters` module defines the following configuration parameters:

Name	Description
counters.counter_age_limit	The maximum age limit in seconds for a pending counter increment to stay in the internal table. When a pending counter reaches this age limit, the pending counter increments are flushed and the table is cleared. The default is 5 minutes.
	300
counters.counter_limit	The maximum number of different counters which can be stored in the internal table before flushing the pending increments to the database. When this limit is reached, the pending counter increments are flushed and the table is cleared.
	1000

13.3 Counter Declaration

Each counter must be declared by instantiating the `Definition` package. This instantiation serves as identification of the counter and it defines the database table as well as the column in that table that will hold the total counter. The following definition is used for the read counter of a wiki page. The wiki page table contains a `read_count` column and it will be incremented each time the counter is incremented.

```
with AWA.Counters.Definition;
...
package Read_Counter is
   new AWA.Counters.Definition
      (AWA.Wikis.Models.WIKI_PAGE_TABLE, "read_count");
```

When the database table does not contain any counter column, the column field name is not given and the counter definition is defined as follows:

```
with AWA.Counters.Definition;
...
package Login_Counter is
   new AWA.Counters.Definition (AWA.Users.Models.USER_PAGE_TABLE);
```

Sometimes a counter is not associated with any database entity. Such counters are global and they are assigned a unique name.

```
with AWA.Counters.Definition;
```

```ada
...
package Start_Counter is
   new AWA.Counters.Definition (null, "startup_counter");
```

13.4 Incrementing the counter

Incrementing the counter is done by calling the `Increment` operation. When the counter is associated with a database entity, the entity primary key must be given. The counter is not immediately incremented in the database so that several calls to the `Increment` operation will not trigger a database update.

```ada
with AWA.Counters;
...
AWA.Counters.Increment (Counter => Read_Counter.Counter, Key => Id);
```

A global counter is also incremented by using the `Increment` operation.

```ada
with AWA.Counters;
...
AWA.Counters.Increment (Counter => Start_Counter.Counter);
```

13.5 Ada Bean

The `Counter_Bean` allows to represent a counter associated with some database entity and allows its control by the `<awa:counter>` HTML component. To use it, an instance of the `Counter_Bean` should be defined in a another Ada bean declaration and configured. For example, it may be declared as follows:

```ada
type Wiki_View_Bean is new AWA.Wikis.Models.Wiki_View_Info
with record
  ...
  Counter : aliased Counter_Bean
     (Of_Type => ADO.Objects.KEY_INTEGER,
      Of_Class => AWA.Wikis.Models.WIKI_PAGE_TABLE);
end record;
```

The counter value is held by the `Value` member of `Counter_Bean` and it should be initialized programatically when the Ada bean instance is loaded (for example through a `load` action). The `Counter_Bean` needs to know the database entity to which it is associated and its `Object` member must be initialized. This is necessary for the `<awa:counter>` HTML component to increment the associated counter when the page is displayed. Below is an extract of such initialization:

```ada
procedure Load
  (Bean     : in out Wiki_View_Bean;
   Outcome  : in out Ada.Strings.Unbounded.Unbounded_String) is
begin
  ...
  Bean.Counter.Value := Bean.Get_Read_Count;
  ADO.Objects.Set_Value (Bean.Counter.Object, Bean.Get_Id);
end Load;
```

The `Stat_List_Bean` allows to retrieve the list of counters per day for a given database entity. It needs a special managed bean configuration that describes the database entity type, the counter name and SQL query name.

The example below from the Wikis Module declares the bean `wikiPageStats`. The database entity is `awa_wiki_page` which is the name of the database table that holds wiki page. The SQL query to retrieve the result is `page-access-stats`.

```xml
<description>The counter statistics for a wiki page</description>
<managed-bean-name>wikiPageStats</managed-bean-name>
<managed-bean-class>AWA.Counters.Beans.Stat_List_Bean</managed-bean-
    class>
<managed-bean-scope>request</managed-bean-scope>
<managed-property>
  <property-name>entity_type</property-name>
  <property-class>String</property-class>
  <value>awa_wiki_page</value>
</managed-property>
<managed-property>
  <property-name>counter_name</property-name>
  <property-class>String</property-class>
  <value>read_count</value>
</managed-property>
<managed-property>
  <property-name>query_name</property-name>
  <property-class>String</property-class>
  <value>page-access-stats</value>
</managed-property>
 </managed-bean>
```

A typical XHTML view that wants to use such bean, should call the `load` action at beginning to load the counter statistics by running the SQL query.

```
<f:view contentType="application/json; charset=UTF-8"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html">
  <f:metadata>
    <f:viewAction action='#{wikiPageStats.load}'/>
  </f:metadata>
{"data":[<h:list value="#{wikiPageStats.stats}"
  var="stat">["#{stat.date}", #{stat.count}],</h:list>[0,0]]}
</f:view>
```

13.6 HTML components

The `<awa:counter>` component is an Ada Server Faces component that allows to increment and display easily the counter. The component works by using the `Counter_Bean` Ada bean object which describes the counter in terms of counter definition, the associated database entity, and the current counter value.

```
<awa:counter value="#{wikiPage.counter}"/>
```

When the component is included in a page the `Counter_Bean` instance associated with the EL `value` attribute is used to increment the counter. This is similar to calling the `AWA.Counters.Increment` operation from the Ada code.

13.7 Data model

The `counters` module has a simple database model which needs two tables. The `Counter_Definition` table is used to keep track of the different counters used by the application. A row in that table is created for each counter declared by instantiating the `Definition` package. The `Counter` table holds the counters for each database entity and for each day. By looking at that table, it becomes possible to look at the daily access or usage of the counter.

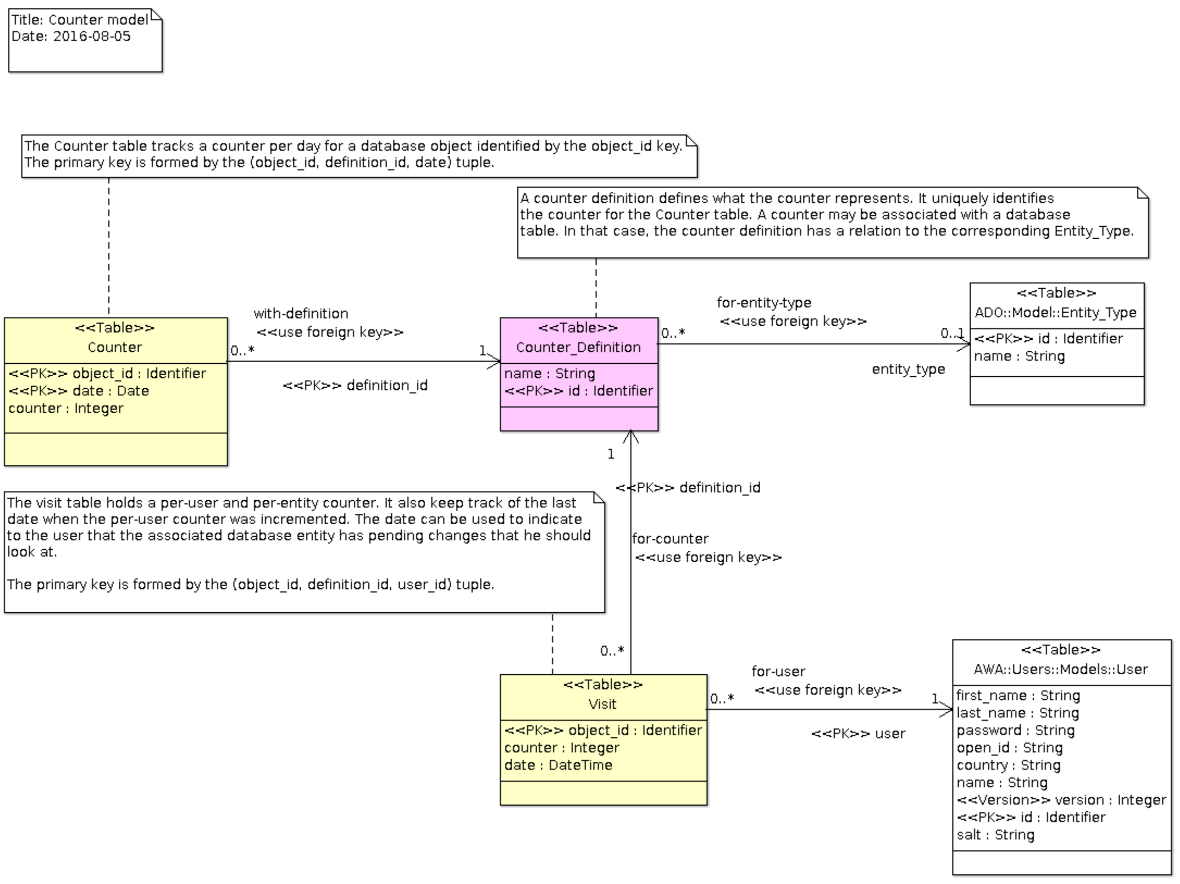

Title: Counter model
Date: 2016-08-05
The Counter table tracks a counter per day for a database object identified by the object_id key.
The primary key is formed by the (object_id, definition_id, date) tuple.
A counter definition defines what the counter represents. It uniquely identifies the counter for the Counter table. A counter may be associated with a database table. In that case, the counter definition has a relation to the corresponding Entity_Type.
<<Table>>
Counter
<<PK>> object_id : Identifier
<<PK>> date : Date
counter : Integer
with-definition
<<use foreign key>>
0..*
1
<<PK>> definition_id
<<Table>>
Counter_Definition
name : String
<<PK>> id : Identifier
for-entity-type
<<use foreign key>>
0..*
0..1
entity_type
<<Table>>
ADO::Model::Entity_Type
<<PK>> id : Identifier
name : String
1
<<PK>> definition_id
for-counter
<<use foreign key>>
0..*
The visit table holds a per-user and per-entity counter. It also keep track of the last date when the per-user counter was incremented. The date can be used to indicate to the user that the associated database entity has pending changes that he should look at.
The primary key is formed by the (object_id, definition_id, user_id) tuple.
<<Table>>
Visit
<<PK>> object_id : Identifier
counter : Integer
date : DateTime
for-user
<<use foreign key>>
0..*
1
<<PK>> user
<<Table>>
AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String

Figure 21

14 Votes Module

The `votes` module allows users to vote for objects defined in the application. Users can vote by setting a rating value on an item (+1, -1 or any other integer value). The `votes` module makes sure that users can vote only once for an item. A global rating is associated with the item to give the vote summary. The vote can be associated with any database entity and it is not necessary to change other entities in your data model.

14.1 Integration

To be able to use the `votes` module, you will need to add the following line in your GNAT project file:

```
with "awa_votes";
```

The `Vote_Module` manages the votes on entities. It provides operations that are used by the vote beans or other services to vote for an item. An instance of the `Vote_Module` must be declared and registered in the AWA application.

The module instance can be defined as follows:

```
type Application is new AWA.Applications.Application with record
   Vote_Module : aliased AWA.Votes.Modules.Vote_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App    => App.Self.all'Access,
          Name   => AWA.Votes.Modules.NAME,
          URI    => "votes",
          Module => App.Vote_Module'Access);
```

14.2 Ada Beans

The `Vote_Bean` is a bean intended to be used in presentation files (XHTML facelet files) to vote for an item. The managed bean can be easily configured in the application XML configuration file. The `permission` and `entity_type` are the two properties that should be defined in the configuration. The `permission` is the name of the permission that must be used to verify that the user is allowed to vote for the item. The `entity_type` is the name of the entity (table name) used by the item. The example below defines the bean `questionVote` defined by the question module.

```
<managed-bean>
  <description>The vote bean that allows to vote for a question.</
      description>
  <managed-bean-name>questionVote</managed-bean-name>
  <managed-bean-class>AWA.Votes.Beans.Votes_Bean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>permission</property-name>
    <property-class>String</property-class>
    <value>answer-create</value>
  </managed-property>
  <managed-property>
    <property-name>entity_type</property-name>
    <property-class>String</property-class>
    <value>awa_question</value>
  </managed-property>
</managed-bean>
```

The vote concerns entities for the `awa_question` entity table. The permission `answer-create` is used to verify that the vote is allowed.

The managed bean defines three operations that can be called: `vote_up`, `vote_down` and `vote` to setup specific ratings.

14.3 Javascript integration

The `votes` module provides a Javascript support to help users vote for items. The Javascript file `/js/awa-votes.js` must be included in the Javascript page. It is based on jQuery and ASF. The vote actions are activated on the page items as follows in XHTML facelet files:

```
<util:script>
  $('.question-vote').votes({
    voteUrl: "#{contextPath}/questions/ajax/questionVote/vote?id=",
    itemPrefix: "vote_for-"
});
</util:script>
```

When the vote up or down HTML element is clicked, the `vote` operation of the managed bean `questionVote` is called. The operation will update the user's vote for the selected item (in the example "a question").

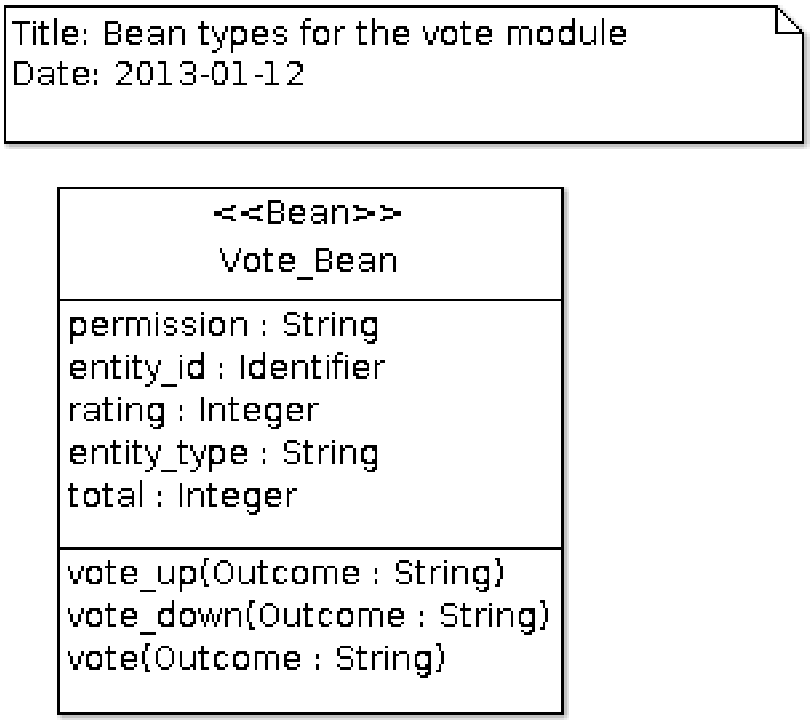

Title: Bean types for the vote module
Date: 2013-01-12
<<Bean>>
Vote_Bean
permission : String
entity_id : Identifier
rating : Integer
entity_type : String
total : Integer
vote_up(Outcome : String)
vote_down(Outcome : String)
vote(Outcome : String)

Figure 22

14.4 Data model

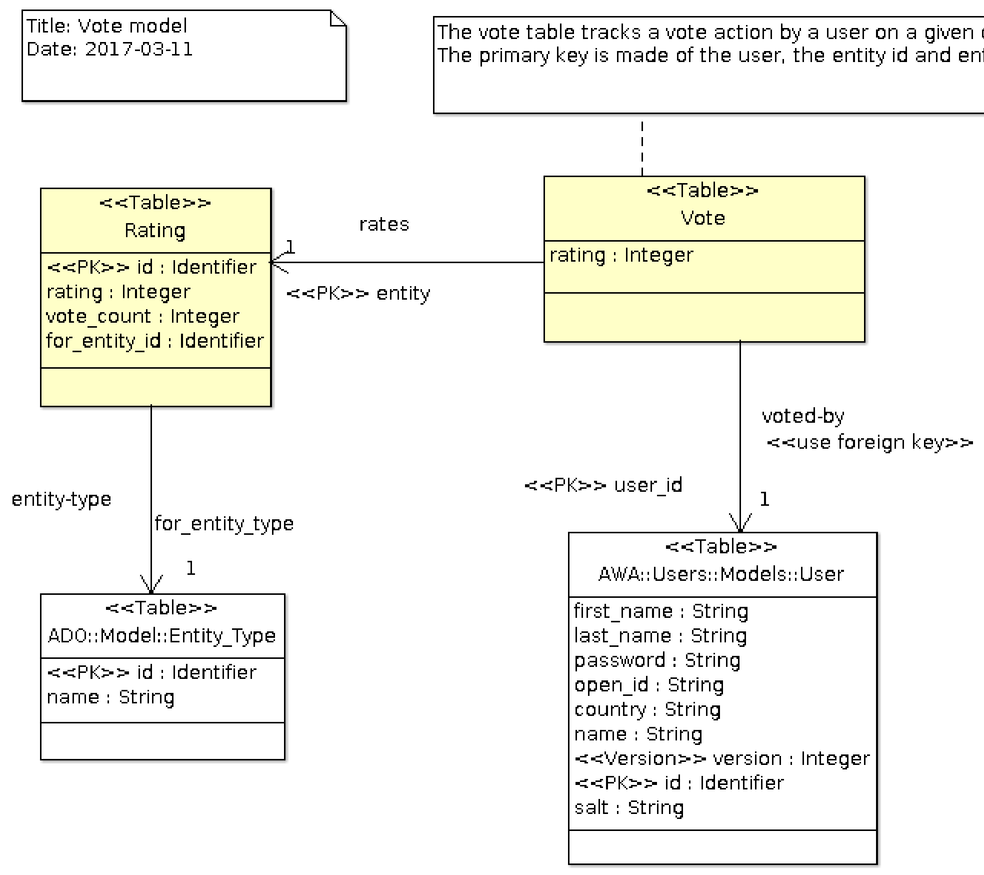


Figure 23

15 Tags Module

The `Tags` module allows to associate general purpose tags to any database entity. It provides a JSF component that allows to insert easily a list of tags in a page and in a form. An application can use the bean types defined in `AWA.Tags.Beans` to define the tags and it will use the `awa:tagList` component to display them. A tag cloud is also provided by the `awa:tagCloud` component.

15.1 Integration

The Tag_Module manages the tags associated with entities. It provides operations that are used by the tag beans together with the awa:tagList and awa:tagCloud components to manage the tags. An instance of the Tag_Module must be declared and registered in the AWA application. The module instance can be defined as follows:

```
type Application is new AWA.Applications.Application with record
   Tag_Module : aliased AWA.Tags.Modules.Tag_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App     => App.Self.all'Access,
          Name    => AWA.Tags.Modules.NAME,
          URI     => "tags",
          Module  => App.Tag_Module'Access);
```

15.2 Ada Beans

Several bean types are provided to represent and manage a list of tags. The tag module registers the bean constructors when it is initialized. To use them, one must declare a bean definition in the application XML configuration.

15.2.1 Tag_List_Bean

The `Tag_List_Bean` holds a list of tags and provides operations used by the `awa:tagList` component to add or remove tags within a `h:form` component. A bean can be declared and configured as follows in the XML application configuration file:

```
<managed-bean>
  <managed-bean-name>questionTags</managed-bean-name>
```

```xml
  <managed-bean-class>AWA.Tags.Beans.Tag_List_Bean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>entity_type</property-name>
    <property-class>String</property-class>
    <value>awa_question</value>
  </managed-property>
  <managed-property>
    <property-name>permission</property-name>
    <property-class>String</property-class>
    <value>question-edit</value>
  </managed-property>
</managed-bean>
```

The `entity_type` property defines the name of the database table to which the tags are assigned. The `permission` property defines the permission name that must be used to verify that the user has the permission do add or remove the tag. Such permission is verified only when the `awa:tagList` component is used within a form.

15.2.2 Tag_Search_Bean

The `Tag_Search_Bean` is dedicated to searching for tags that start with a given pattern. The auto complete feature of the `awa:tagList` component can use this bean type to look in the database for tags matching a start pattern. The declaration of the bean should define the database table to search for tags associated with a given database table. This is done in the XML configuration with the `entity_type` property.

```xml
<managed-bean>
  <managed-bean-name>questionTagSearch</managed-bean-name>
  <managed-bean-class>AWA.Tags.Beans.Tag_Search_Bean</managed-bean-
     class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>entity_type</property-name>
    <property-class>String</property-class>
    <value>awa_question</value>
  </managed-property>
</managed-bean>
```

15.2.3 Tag_Info_List_Bean

The Tag_Info_List_Bean holds a collection of tags with their weight. It is used by the awa:tagCloud component.

```xml
<managed-bean>
  <managed-bean-name>questionTagList</managed-bean-name>
  <managed-bean-class>AWA.Tags.Beans.Tag_Info_List_Bean</managed-bean-
      class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>entity_type</property-name>
    <property-class>String</property-class>
    <value>awa_question</value>
  </managed-property>
</managed-bean>
```

AWA.Tags.Models.Tag_Info

The tag information.

Type	Ada	Name	Description
	String	tag	the tag name.
	Natural	count	the number of references for the tag.

15.3 HTML components

15.3.1 Displaying a list of tags

The awa:tagList component displays a list of tags. Each tag can be rendered as a link if the tagLink attribute is defined. The list of tags is passed in the value attribute. When rending that list, the var attribute is used to setup a variable with the tag value. The tagLink attribute is then evaluated against that variable and the result defines the link.

```xml
<awa:tagList value='#{questionList.tags}' id='qtags' styleClass="
    tagedit-list"
             tagLink="#{contextPath}/questions/tagged.html?tag=#{
                 tagName}"
             var="tagName"
```

```
                tagClass="tagedit-listelement tagedit-listelement-old"/>
```

15.3.2 Tag editing

The awa:tagList component allows to add or remove tags associated with a given database entity. The tag management works with the jQuery plugin Tagedit. For this, the page must include the /js/jquery.tagedit.js Javascript resource.

The tag edition is active only if the awa:tagList component is placed within an h:form component. The value attribute defines the list of tags. This must be a Tag_List_Bean object.

```
<awa:tagList value='#{question.tags}' id='qtags'
             autoCompleteUrl='#{contextPath}/questions/lists/tag-search
                 .html'/>
```

When the form is submitted and validated, the procedure Set_Added and Set_Deleted are called on the value bean with the list of tags that were added and removed. These operations are called in the UPDATE_MODEL_VALUES phase (ie, before calling the action’s bean operation).

15.3.3 Tag cloud

The awa:tagCloud component displays a list of tags as a tag cloud. The tags list passed in the value attribute must inherit from the Tag_Info_List_Bean type which indicates for each tag the number of times it is used.

```
<awa:tagCloud value='#{questionTagList}' id='cloud' styleClass="tag-
    cloud"
              var="tagName" rows="30"
              tagLink="#{contextPath}/questions/tagged.html?tag=#{
                  tagName}"
              tagClass="tag-link"/>
```

15.4 Queries

Name	Description
check-tag	Check and get the tag identifier associated with a given tag and entity
tag-list	Get the list of tags associated with a given database entity

Name	Description
tag-search	Get the list of tag names that match some string
tag-list-all	Get the list of tags associated with all the database entities of a given type
tag-list-for-entities	Get the list of tags associated with a set of entities of the same type.

15.5 Data model

The database model is generic and it uses the `Entity_Type` provided by Ada Database Objects to associate a tag to entities stored in different tables. The `Entity_Type` identifies the database table and the stored identifier in `for_entity_id` defines the entity in that table.

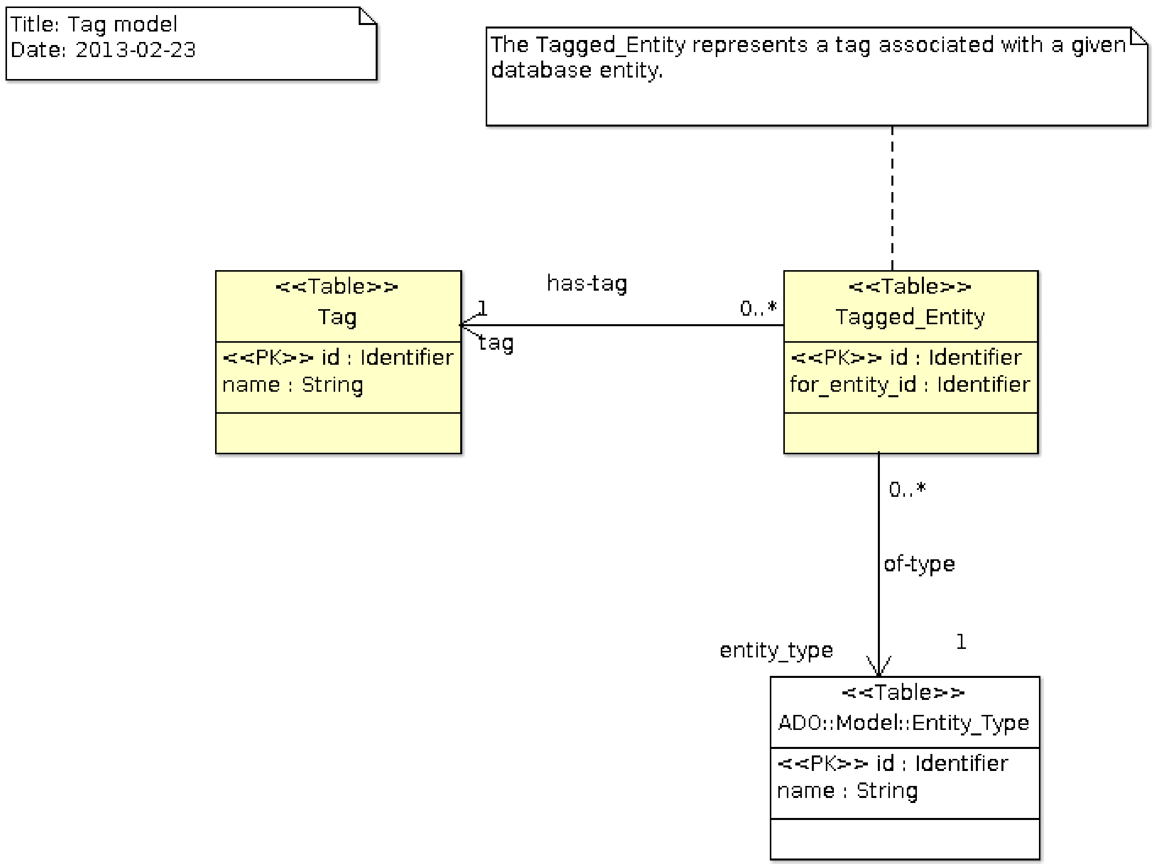


Figure 24

16 Comments Module

The `Comments` module is a general purpose module that allows to associate user comments to any database entity. The module defines several bean types that allow to display a list of comments or edit and publish a new comment.

16.1 Integration

The Comment_Module manages the comments associated with entities. It provides operations that are used by the comment beans to manage the comments. An instance of the Comment_Module must be declared and registered in the AWA application. The module instance can be defined as follows:

```ada
type Application is new AWA.Applications.Application with record
   Comment_Module : aliased AWA.Comments.Modules.Comment_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```ada
Register (App     => App.Self.all'Access,
          Name    => AWA.Comments.Modules.NAME,
          URI     => "comments",
          Module  => App.Comment_Module'Access);
```

16.2 Ada Beans

Several bean types are provided to represent and manage a list of tags. The tag module registers the bean constructors when it is initialized. To use them, one must declare a bean definition in the application XML configuration.

16.2.1 Comment_List_Bean

The `Comment_List_Bean` holds a list of comments and provides operations used by the `awa:tagList` component to add or remove tags within a `h:form` component. A bean can be declared and configured as follows in the XML application configuration file:

```xml
<managed-bean>
  <managed-bean-name>postCommentList</managed-bean-name>
  <managed-bean-class>AWA.Comments.Beans.Comment_List_Bean</managed-
      bean-class>
```

```
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
      <property-name>entity_type</property-name>
      <property-class>String</property-class>
      <value>awa_post</value>
    </managed-property>
    <managed-property>
      <property-name>permission</property-name>
      <property-class>String</property-class>
      <value>blog-comment-post</value>
    </managed-property>
    <managed-property>
      <property-name>sort</property-name>
      <property-class>String</property-class>
      <value>oldest</value>
    </managed-property>
    <managed-property>
      <property-name>status</property-name>
      <property-class>String</property-class>
      <value>published</value>
    </managed-property>
  </managed-bean>
```

The `entity_type` property defines the name of the database table to which the comments are assigned. The `permission` property defines the permission name that must be used to verify that the user has the permission do add or remove the comment.

AWA.Comments.Models.Comment_Info

The comment information.

Type	Ada	Name	Description
	Identifier	id	the comment identifier.
	String	author	the comment author's name.
	String	email	the comment author's email.
	Date	date	the comment date.
	AWA.Comments.Models.Format_Type	format	the comment format type.
	String	comment	the comment text.

Type	Ada	Name	Description
	AWA.Comments.Models.Status_Type	status	the comment status.

Name	Description
comment-list	Get the list of comments associated with given database entity
all-comment-list	Get the list of comments associated with given database entity

16.3 Data model

The database model is generic and it uses the `Entity_Type` provided by Ada Database Objects to associate a comment to entities stored in different tables. The `Entity_Type` identifies the database table and the stored identifier in `for_entity_id` defines the entity in that table.

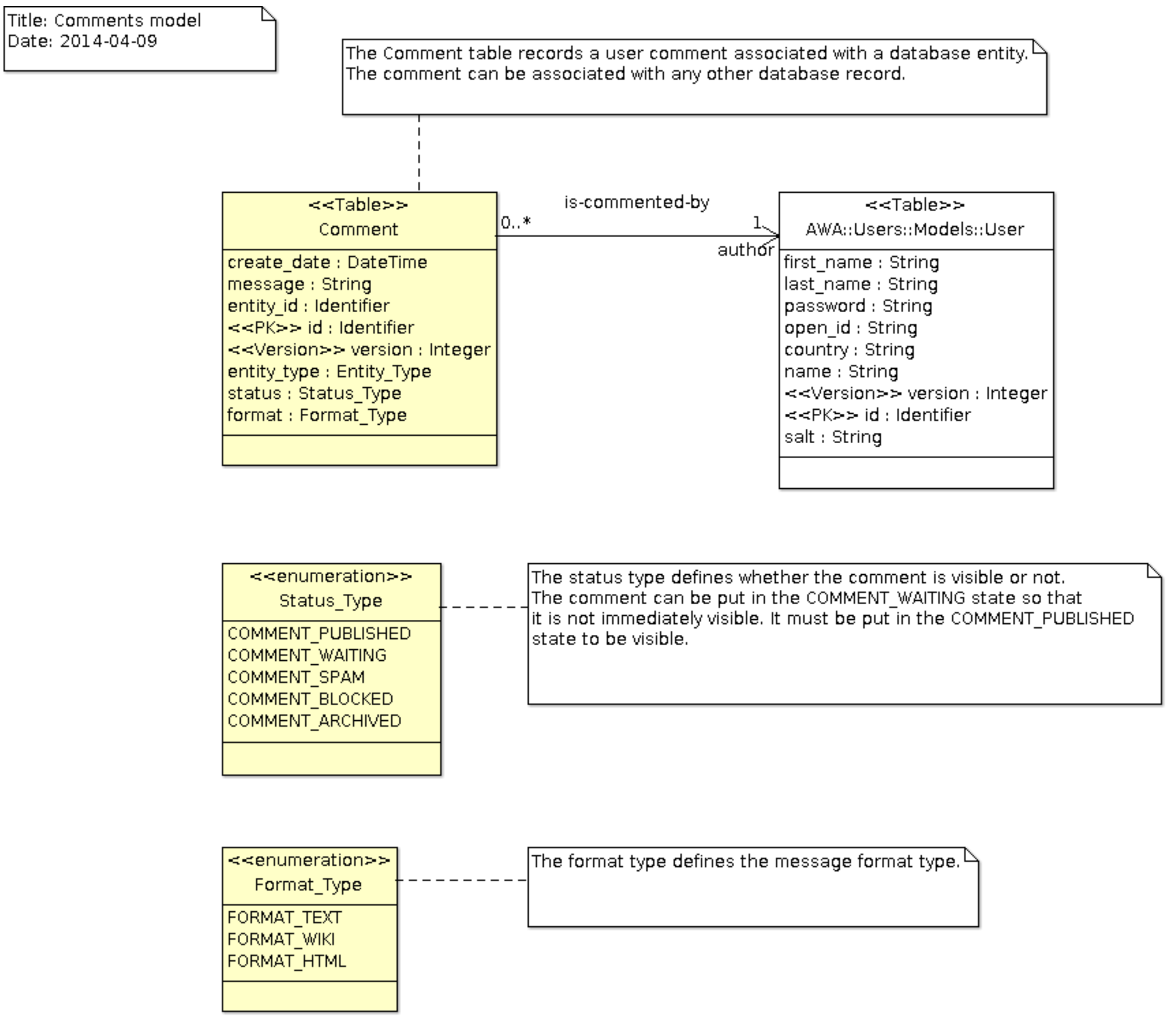

Title: Comments model
Date: 2014-04-09
The Comment table records a user comment associated with a database entity.
The comment can be associated with any other database record.
<<Table>>
Comment
create_date : DateTime
message : String
entity_id : Identifier
<<PK>> id : Identifier
<<Version>> version : Integer
entity_type : Entity_Type
status : Status_Type
format : Format_Type
is-commented-by
0..*
1
author
<<Table>>
AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
<<enumeration>>
Status_Type
COMMENT_PUBLISHED
COMMENT_WAITING
COMMENT_SPAM
COMMENT_BLOCKED
COMMENT_ARCHIVED
The status type defines whether the comment is visible or not.
The comment can be put in the COMMENT_WAITING state so that
it is not immediately visible. It must be put in the COMMENT_PUBLISHED
state to be visible.
<<enumeration>>
Format_Type
FORMAT_TEXT
FORMAT_WIKI
FORMAT_HTML
The format type defines the message format type.

Figure 25

17 Settings Module

The `Settings` module provides management of application and user settings. A setting is identified by a unique name in the application. It is saved in the database and associated with a user.

17.1 Getting a user setting

Getting a user setting is as simple as calling a function with the setting name and the default value. If the setting was modified by the user and saved in the database, the saved value will be returned. Otherwise, the default value is returned. For example, if an application defines a `row-per-page` setting to define how many rows are defined in a list, the user setting can be retrieved with:

```
Row_Per_Page : constant Integer := AWA.Settings.Get_User_Setting ("row-
    per-page", 10);
```

17.2 Saving a user setting

When a user changes the setting value, we just have to save it in the database. The setting value will either be updated if it exists or created.

```
AWA.Settings.Set_User_Setting ("row-per-page", 20);
```

17.3 Integration

The `Setting_Module` manages the application and user settings. An instance of the the `Setting_Module` must be declared and registered in the AWA application. The module instance can be defined as follows:

```
type Application is new AWA.Applications.Application with record
   Setting_Module : aliased AWA.Settings.Modules.Setting_Module;
end record;
```

And registered in the `Initialize_Modules` procedure by using:

```
Register (App    => App.Self.all'Access,
          Name   => AWA.Settings.Modules.NAME,
          URI    => "settings",
          Module => App.Setting_Module'Access);
```

17.4 Data model

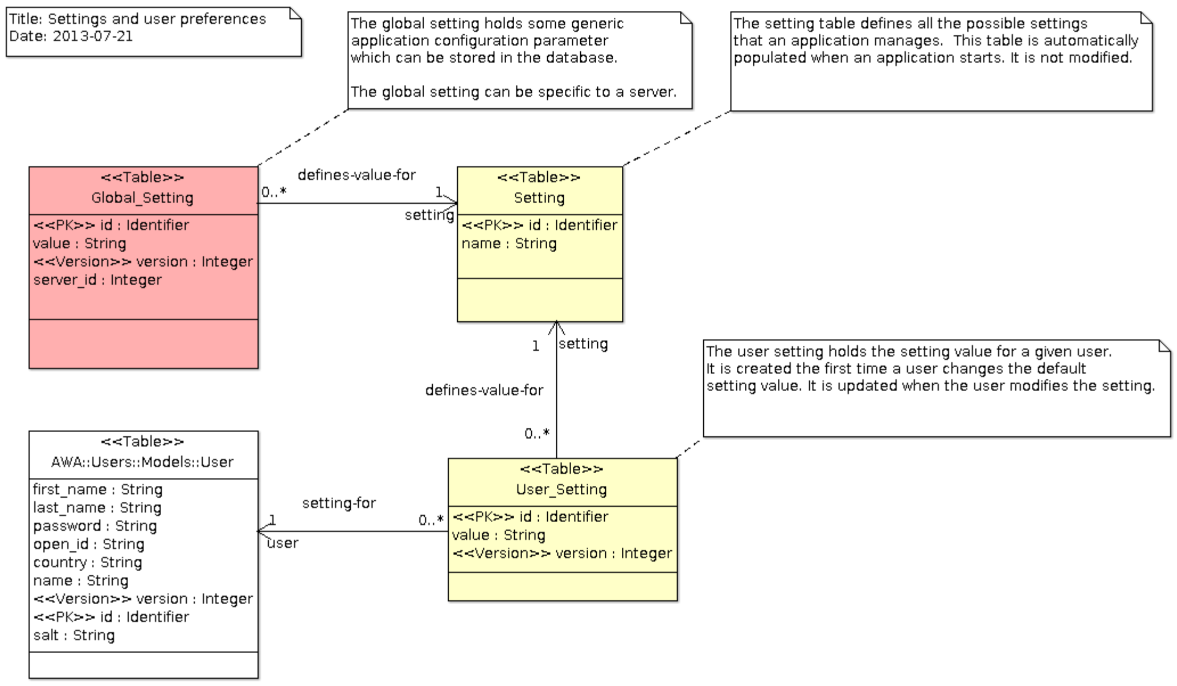

Title: Settings and user preferences
Date: 2013-07-21
The global setting holds some generic application configuration parameter which can be stored in the database.
The global setting can be specific to a server.
The setting table defines all the possible settings that an application manages. This table is automatically populated when an application starts. It is not modified.
<<Table>> Global_Setting
<<PK>> id : Identifier
value : String
<<Version>> version : Integer
server_id : Integer
defines-value-for
0..*
1
setting
<<Table>> Setting
<<PK>> id : Identifier
name : String
1
setting
defines-value-for
0..*
The user setting holds the setting value for a given user. It is created the first time a user changes the default setting value. It is updated when the user modifies the setting.
<<Table>> AWA::Users::Models::User
first_name : String
last_name : String
password : String
open_id : String
country : String
name : String
<<Version>> version : Integer
<<PK>> id : Identifier
salt : String
setting-for
1
user
0..*
<<Table>> User_Setting
<<PK>> id : Identifier
value : String
<<Version>> version : Integer

Figure 26

18 Setup Application

The AWA.Setup package implements a simple setup application that allows to configure the database, the Google and Facebook application identifiers and some other configuration parameters. It is intended to help in the installation process of any AWA-based application.

It defines a specific web application that is installed in the web container for the duration of the setup. The setup application takes control over all the web requests during the lifetime of the installation. As soon as the installation is finished, the normal application is configured and installed in the web container and the user is automatically redirected to it.

18.1 Integration

To be able to use the setup application, you will need to add the following line in your GNAT project file:

```
with "awa_setup";
```

The setup application can be integrated as an AWA command by instantiating the `AWA.Commands.Setup` generic package. To integrate the `setup` command, you will do:

```
with AWA.Commands.Start;
with AWA.Commands.Setup;
...
package Start_Command is new AWA.Commands.Start (Server_Commands);
package Setup_Command is new AWA.Commands.Setup (Start_Command);
```

18.2 Setup Procedure Instantiation

The setup process is managed by the *Configure* generic procedure. The procedure must be instantiated with the application class type and the application initialize procedure.

```
 procedure Setup is
    new AWA.Setup.Applications.Configure (MyApp.Application'Class,
                                          MyApp.Application_Access,
                                          MyApp.Initialize);
```

18.3 Setup Operation

The *Setup* instantiated operation must then be called with the web container. The web container is started first and the *Setup* procedure gets as parameter the web container, the application instance to configure, the application name and the application context path.

```
Setup (WS, App, "atlas", MyApp.CONTEXT_PATH)
```

The operation will install the setup application to handle the setup actions. Through the setup actions, the installer will be able to:

- Configure the database (MySQL or SQLite),
- Configure the Google+ and Facebook OAuth authentication keys,
- Configure the application name,
- Configure the mail parameters to be able to send email.

After the setup and configure is finished, the file .initialized is created in the application directory to indicate the application is configured. The next time the *Setup* operation is called, the installation process will be skipped.

To run again the installation, remove manually the .initialized file.

19 Tips

19.1 UI Presentation Tips

19.1.1 Adding a simple page

To add a new presentation page in the application, you can use the Dynamo code generator. The web page is an XHTML file created under the `web` directory. The page name can contain a directory that will be created if necessary. The new web page can be configured to use a given layout. The layout file must exist to be used. The default layout is `layout`. You can create a new layout with `add-layout` command. You can also write your layout by adding an XHTML file in the directory:

```
web/WEB-INF/layouts
```

To create the new web page `web/todo/list.xhtml`, you will use:

```
dynamo add-page todo/list
```

Depending on your application configuration and the URL used by the new page, you may have to add or modify a `url-policy`. By default, if the new URL does not match an existing `url-policy`, the access will be denied for security reasons. To allow anonymous users to access the page, use the following `url-policy`:

```
<url-policy>
  <permission>anonymous</permission>
  <url-pattern>/todo/list.html</url-pattern>
</url-policy>
```

and if you want only logged users, use the following:

```
<url-policy>
  <permission>logged-user</permission>
  <url-pattern>/todo/list.html</url-pattern>
</url-policy>
```

Note: make sure to replace the `.xhtml` extension by `.html`.

19.1.2 Add Open Graph

Use a combination of `fn:trim`, `fn:substring` and `util:escapeJavaScript` to create the Open Graph description. The first two functions will remove spaces at begining and end of the description and will truncate the string. The `util:escapeJavaScript` is then necessary to make a value HTML attribute when the description contains special characters.

```
<meta property="og:description"
 content="#{util:escapeJavaScript(fn:substring(fn:trim(post.description
    ),1,128))}"/>
```

19.2 Configuration Tips

19.2.1 Adding a permission on user creation

Sometimes it is useful to add some permission when a user is created. This can be done programatically but also through some simple configuration. By using the `on-event` XML definition, it is possible to create a set of permissions when the `user-create` event is posted, hence during user creation.

The following XML extract from Atlas demonstrator will add several Wiki permissions to the new user.

```
<on-event name="user-create">
  <action>#{permission.create}</action>
  <property name="entity_type">awa_wiki_space</property>
  <property name="entity_id">1</property>
  <property name="workspace_id">1</property>
  <property name="permission">wiki-page-create,wiki-page-update,wiki-
      page-delete,wiki-page-view,wiki-space-delete,wiki-space-update</
      property>
</on-event>
```

19.2.2 Secure configuration

Setting up the secure configuration is made by using a secure keystore with the `akt` tool. The secure configuration is stored in the keystore file that `akt` will protect by encrypting each configuration property with their own encryption key. In order to setup and use the secure configuration the following steps are necessary:

- Create the keystore file and protect it with a password or a GPG key,
- Populate the keystore file with the configuration values,
- Launch the server with specific options in order to use and access the keystore file.

The two initial steps are done by using the `akt` tool.

To create the keystore file, one way is to run the `akt` command and give it the keystore password:

```
mkdir secure
akt create --wallet-key-file=secure/wallet.key -c 100000:300000 \
  secure/config.akt
```

This will generate the `secure/wallet.key` file and setup the keystore file in `secure/config.akt`. The password must be given to the server when it is started. To avoid that, it is possible to store it in a file and make sure the file is protected against read and write access. If the password is stored in such file, the keystore is created by using:

```
akt create --wallet-key-file=secure/wallet.key \
  --passfile=secure/master.key -c 100000:300000 \
  secure/config.akt
```

Once the keystore is created, the configuration are inserted. Because the server command can use only one keystore and have several applications, the configuration parameter must be prefixed by the application name. For example, to setup the `database` configuration for the `atlas` application, you will use the command:

```
akt set --wallet-key-file=secure/wallet.key \
  --passfile=secure/master.key \
  secure/config.akt \
  atlas.database 'mysql://localhost:3306/atlas?user=atlas&password=
      PiX2ShaimohW6eno
```

To avoid having to specify several configuration parameters when launching the server, it is good practice to create a server global configuration file and indicate several parameters that the server will use. Create a file `secure/config.properties` that contains:

```
keystore-path=secure/config.akt
keystore-masterkey-path=secure/master.key
keystore-password-path=secure/password.key
```

Then, to start the server we just need to give it the server global configuration path:

```
bin/atlas-server -c secure/config.properties start
```

Note that in order to use this configuration setup, the directory must have the `rwx------` rights and files must have the `rwx------` rights.