Nibiru Reference

October 21, 2011

http://code.google.com/p/nibiru/

Part I

Introduction

1 Framework objective

The framework objective is to facilitate the building of modular applications. The following goals are established in order to meet such objective:

- Providing an abstraction layer over different technologies in order to avoid coupling.
- Providing services which are common to business applications, such as CRUDs, reports, workflow, transaction management, security and internationalization.
- Providing dynamic update mechanisms for the system in order to allow hot swapping.
- Implementing patterns which facilitate solving problems in a structured way. But avoiding to force the user to implement a given solution.
- Facilitate decoupled communication among modules.
- Avoiding reinvent the wheel. Creating layers of abstraction butusing existing technologies when possible.

2 Architecture

This section explains architectural decisions.

2.1 IoC pattern

In order to decouple each component from the container and other components, the dependencies of each component are injected (IoC pattern).

2.2 MVP pattern

The model used for the presentation layer is the MVP pattern, under its passive view variant. This allows the presenters to be decoupled from each other by an event bus and also to be decoupled from view implementation. Google also makes a good description of this pattern.

Also, the concept of abstracting the view was taken a step further, creating abstractions for common components. Thus, the user can choose creating a generic view or creating a view using the particular advantages of a specific technology.

2.3 API / implementation separation

We define two kind of modules, in order to facilitate the decoupling among different modules implementations:

- API: Contains interfaces to be exposed to other components. By convention the name ends with ".api".
- Implementation: Contains API implementations. By convention the names are almost equals to the implemented API name, but changing ".api" suffix by something descriptive of the implementation.

In general, any module can only access another module through an API. The exception to this rule are modules with utility classes that do not expose services.

Another naming convention is that implementations of APIs that are not dependent on a particular technology will have a ".generic" suffix.

2.4 Extension points

The system has an extension point mechanism for adding or removing functionality dynamically. The idea was taken from Eclipse platform, but trying to take a simpler approach.

2.5 Extension using scripting

The framework allows customization through scripting. This, combined with the extension point mechanism allows the user to add functionality to the system without the need to compile, package or install anything. The same application can provide a module for administering such extra extensions.

2.6 Java platform

Javawas chosen because it is currently the most widespread platform within the enterprise applications, in addition to being easily portable to different environments and having many frameworks and libraries.

2.7 OSGi / Spring DM

We chose OSGi because it provides a mechanism for dynamic module management. Spring DM is used because it provides many facilities to implement the IoC pattern under OSGi. Gemini wasn't chosen because at the time of starting the project it was still very immature.

Using these technologies, shared components are exposed using OSGi services. Also, the division between API and implementation allows service hot swapping, since the client components doesn't access to the concrete class implementation. On the other hand, Spring DM provides proxies that make such hot swapping transparent to the client code.

However, almost all components are independent of OSGi and Spring, thanks to the IoC pattern (except for the ones that implement specific Spring features).

3 Getting started

3.1 Required software

1. Java (http://www.java.com/en/download/).
2. Eclipse (http://www.eclipse.org/).
3. Maven (http://maven.apache.org/).
4. A GIT client (http://git-scm.com/). We use EGit.

3.2 Installation

1. Clone the project as explained in http://code.google.com/p/nibiru/source/checkout
2. Run “mvn eclipse:eclipse” from root directory in order to build the Eclipse project from Maven files and downloading target platform JARS.
3. Go to ar.com.oxen.sample/ar.com.oxen.sample.targetplatform and run “mvn compile” in order to build the target platform from Maven dependencies.
4. Import the projects into Eclipse.
5. At preferences menu, activate the Nibiru target platform. Select “reload” option in order to recognize the downloaded JARs.
6. Run the OSGi application launch called "Nibiru Test". By default, Eclipse adds all the plugin (OSGi) projects which are open in the workspace, so even if there is a JAR with the same module, the source project takes precedence.

If you don’t want to download the full source code, you can run it downloading a precompiled sample app: http://nibiru.googlecode.com/files/sampleapp.zip. Although the sample app is packaged, you must build the target platform and run it from Eclipse.

3.3 Sample project

Running the sample application will create an H2 database in a directory called nibiruDb inside your home directory. Windows users should modify the ar.com.oxen.nibiru.sample/ar.com.oxen.nibiru.sample.datasource.fragment/src/main/resources/database.prop
file in order to specify the database location.

The sample application uses a dummy authentication service. Log-in with user “guest”, password “guest”.

TODO: Simplificar el armado de un proyecto. Opciones:

1. *Hacer un namespace handler.*
2. *Armar anotaciones (aunque no se cómo encajaría esto con Spring DM).*
3. *Usar directamente Guice+Peaberry (http://code.google.com/p/peaberry/)*

Part II

Modules

4 Base application

The ar.com.oxen.nibiru.application.api bundle contains interfaces used to implement basic functions such as application login, "about" window , etc.

The idea is that an implementation of this bundle must provide the basis to setput the application. All the extra functionality will be added by other modules.

This module contains factories for presenters:

```
package ar.com.oxen.nibiru.application.api;

import ar.com.oxen.nibiru.application.api.about.AboutView;
import ar.com.oxen.nibiru.application.api.login.LoginView;
import ar.com.oxen.nibiru.application.api.main.MainView;
import ar.com.oxen.nibiru.ui.api.mvp.Presenter;

/**
 * Presenter factory for common application functionality.
 */
public interface ApplicationPresenterFactory {
        /**
         * Builds the presenter for login window.
         *
         * @return The presenter
         */
        Presenter<LoginView> buildLoginPresenter();

        /**
         * Builds the presenter for main window.
         *
         * @return The presenter
         */
        Presenter<MainView> buildMainPresenter();

        /**
         * Builds the presenter for about window.
         *
         * @return The presenter
         */
        Presenter<AboutView> buildAboutPresenter();
}
```

and for application views:

```
package ar.com.oxen.nibiru.application.api;

import ar.com.oxen.nibiru.application.api.about.AboutView;
import ar.com.oxen.nibiru.application.api.login.LoginView;
import ar.com.oxen.nibiru.application.api.main.MainView;

/**
 * View factory for common application functionality.
 */
public interface ApplicationViewFactory {
        /**
         * Builds the view for login window.
         *
         * @return The view
         */
        LoginView buildLoginView();

        /**
         * Builds the view for main window.
         *
         * @return The view
         */
        MainView buildMainView();

        /**
         * Builds the view for about window.
         *
         * @return The view
         */
        AboutView buildAboutView();
}
```

TODO: Los presentadores y las vistas deberían ir en módulos separados.

The ar.com.oxen.nibiru.application.generic bundle provides a generic implementation of basic application components.

5 Extension points

Interfaces for extension points are found in the ar.com.oxen.nibiru.extensionpoint.api bundle. The design is simple: each extension point has just an interface and a name. Besides, the extensions can be enabled or disabled at runtime.

To perform an action whenever an extension is added or removed, the ExtensionTracker interface must be used :

```
package ar.com.oxen.nibiru.extensionpoint.api;

/**
 * Callback for tracking extension status.
 *
 * @param <T>
 *            The extension type
 */
public interface ExtensionTracker<T> {
        /**
         * Callback method called when a new extension is registered.
         *
         * @param extension
         *            The extension
         */
        void onRegister(T extension);

        /**
         * Callback method called when an existing extension is unregistered.
         *
         * @param extension
         *            The extension
         */
        void onUnregister(T extension);
}
```

which provides the necessary callbacks for those events. The ExtensionTrackers must be registered with the ExtensionPointManager service:

```
package ar.com.oxen.nibiru.extensionpoint.api;

/**
 * Service for managing extensions.
 */
public interface ExtensionPointManager {
        /**
         * Registers an extension under a name and an interface
         *
         * @param <K>
         *            The extension point interface
         * @param extension
         *            The extension
         * @param extensionPointName
         *            The extension point name
         * @param extensionPointInterface
         *            The extension point interface
         */
```

```
    <K> void registerExtension(K extension, String extensionPointName,
                        Class<K> extensionPointInterface);

    /**
     * Un-registers an extension.
     *
     * @param extension
     *              The extension.
     */
    void unregisterExtension(Object extension);

    /**
     * Registers a tracker for a given extension type and name.
     *
     * @param <T>
     *              The type parametrized on the tracker
     * @param <K>
     *              The extension point interface
     * @param tracker
     *              The tracker
     * @param extensionPointName
     *              The extension point name
     * @param extensionPointInterface
     *              The extension point interface
     */
    <T, K extends T> void registerTracker(ExtensionTracker<T> tracker,
                        String extensionPointName, Class<K> extensionPointInterf
}
```

The ExtensionPointManager also provides methods for registering new extensions and unregistering a existing one.

The ar.com.oxen.nibiru.extensionpoint.spring bundle has an extension point implementation based on Spring DM and OSGi services. Under this implementation, each extension point is simply implemented using an OSGi service with a property called "extensionPoint". Such property is used to determine the name of the extension point where functionality should be added.

6 Event bus

Several modules use an event bus. The event bus is accessed using the ar.com.oxen.commons.eventbus.api.Event
interface, which does not belong to Nibiru project but to Oxen Java Commons.
In this project there is also a (pretty) simple implementation of such interface.

7 Modules

As mentioned earlier, the framework is designed so that the functionality can be added as separate modules.

The ar.com.oxen.nibiru.module.utils project provides utility classes for this purpose. Typically, each module will have a component responsible for configuring this module at startup. To that end, this project provides the AbstractModuleConfigurator class, which can be extended in order to create such configurators.

```
package ar.com.oxen.nibiru.module.utils;

import java.util.Collection;
import java.util.LinkedList;

import ar.com.oxen.commons.eventbus.api.EventBus;
import ar.com.oxen.nibiru.extensionpoint.api.ExtensionPointManager;
import ar.com.oxen.nibiru.ui.api.mvp.Presenter;
import ar.com.oxen.nibiru.ui.api.mvp.View;

/**
 * Base class for module configurators.
 *
 * @param <VF>
 *            The view factory class
 * @param <PF>
 *            The presenter factory class
 */
public abstract class AbstractModuleConfigurator<VF, PF> {
        private ExtensionPointManager extensionPointManager;
        private Collection<Object> registeredExtensions = new LinkedList<Object>
        private EventBus eventBus;
        private VF viewFactory;
        private PF presenterFactory;

        /**
         * Starts the module. This method must be externally called (for example
         * with init-method attribute on Spring context XML).
         */
        public void startup() {
                this.eventBus.subscribeAnnotatedObject(this);
                this.configure();
        }

        /**
         * Same as startup, but for shutdown.
         */
```

```
public void shutdown() {
        /* Remove all the extensions */
        for (Object extension : this.registeredExtensions) {
                this.extensionPointManager.unregisterExtension(extension
        }
        this.registeredExtensions.clear();

        this.unconfigure();
}

/**
 * Abstract method to be override in order to customize module
 * configuration.
 */
protected void configure() {
}

/**
 * Abstract method to be override in order to customize module
 * un-configuration.
 */
protected void unconfigure() {
}

/**
 * Activates a view/presenter. Typically this method will be called from
 * subclasses upon the receiving of an event from the bus in order to
 * navigate to a given window.
 *
 * @param <V>
 *              The view type
 * @param view
 *              The view
 * @param presenter
 *              The presenter
 */
protected <V extends View> void activate(V view, Presenter<V> presenter)
        presenter.setView(view);
        presenter.go();
        view.show();
}

/**
 * Registers an extension under a name and an interface. The extension u
 * be automatically un-published when the module will be unloaded.
 *
```

```
     * @param <K>
     *            The extension point interface
     * @param extension
     *            The extension
     * @param extensionPointName
     *            The extension point name
     * @param extensionPointInterface
     *            The extension point interface
     */
    protected <K> void registerExtension(K extension,
                    String extensionPointName, Class<K> extensionPointInterf
            this.extensionPointManager.registerExtension(extension,
                            extensionPointName, extensionPointInterface);
            this.registeredExtensions.add(extension);
    }

    public void setEventBus(EventBus eventBus) {
            this.eventBus = eventBus;
    }

    protected EventBus getEventBus() {
            return eventBus;
    }

    protected VF getViewFactory() {
            return viewFactory;
    }

    public void setViewFactory(VF viewFactory) {
            this.viewFactory = viewFactory;
    }

    protected PF getPresenterFactory() {
            return presenterFactory;
    }

    public void setPresenterFactory(PF presenterFactory) {
            this.presenterFactory = presenterFactory;
    }

    public void setExtensionPointManager(
                    ExtensionPointManager extensionPointManager) {
            this.extensionPointManager = extensionPointManager;
    }
}
```

You should inject all the required dependencies and trigger the startup() method on startup. On shutdown, you should trigger the shutdown() method. In order to provide custom startup/shutdown configuration logic, you can override the configure() and unconfigure() methods.

Typically, this component will set up navigation between different module screens. For this end, the AbstractModuleConfigurator class provides access to the event bus (which must be injected) and sets itself as listener on that bus. So you can add event handling methods annotated with @EventHandler. In order to show a given view/presenter, you can use the activate() method.

Also, the class provides methods for registering extension points (the ExtensionPointManager must be injected). This is helpful, since the extensions are automatically unregistered when the module is down.

Regarding menus, they are implemented via extension points. So it is only necessary to register an extension with the following interface:

```
package ar.com.oxen.nibiru.ui.api.extension;

/**
 * Extension that represents an item on the menu.
 */
public interface MenuItemExtension {
        /**
         * @return The item name
         */
        String getName();

        /**
         * @return The position (lower numbers are shown first)
         */
        int getPosition();

        /**
         * Method to be executed when the menu is created.
         */
        void onClick();
}
```

or with the following one:

```
package ar.com.oxen.nibiru.ui.api.extension;

/**
 * Extension that represents a menu that can contain other menus.
 */
public interface SubMenuExtension {
        /**
```

```
         * @return The sub-menu name
         */
        String getName();

        /**
         * @return The position (lower numbers are shown first)
         */
        int getPosition();

        /**
         * @return The extension point name where entries of this sub-menu shoul
         *         added.
         */
        String getExtensionPoint();
}
```

You must define an extension point name for each menu. The extension point to the main menu is ar.com.oxen.nibiru.menu.

It is worth noting that the ar.com.oxen.nibiru.ui.utils bundle contains simple implementations of these interfaces.

8 Session

Applications usually have some kind of session information. This is, data that are specific to the user that is connected at any given time. Typically, in a Web application, this information is stored in the HTTP session.

To support the goal of keeping the various components decoupled from the implementation, the ar.com.oxen.nibiru.session.api project provides a generic interface for the session.

```
package ar.com.oxen.nibiru.session.api;

/**
 * Component holding session data.
 */
public interface Session {
        /**
         * Gets an object from session data.
         *
         * @param <T>
         *             The object type
         * @param key
         *             The object key (must be unique)
         * @return The object
```

```
     */
    <T> T get(String key);

    /**
     * Puts an object into session data.
     *
     * @param key
     *            The object key (must be unique)
     * @param value
     *            The object
     */
    void put(String key, Object value);

    /**
     * Removes an object from session data.
     *
     * @param key
     *            The object key (must be unique)
     */
    void remove(String key);

    /**
     * @return An String identifying the session.
     */
    String getId();

    /**
     * @return A mutex that can be used in order to synchronize concurrent
     *         (threaded) session access
     */
    Object getMutex();

    /**
     * Registers a listener for session destruction.
     *
     * @param name
     *            The callback name (must be unique)
     * @param callback
     *            The callback
     */
    void registerDestructionCallback(String name, Runnable callback);

    /**
     * @return True if the session is valid
     */
    boolean isValid();
```

}

The ar.com.oxen.nibiru.session.spring.http project provides access to the HTTP session using Spring components (Servlet filters that provide session access through a ThreadLocal).

The ar.com.oxen.nibiru.session.spring.scope project provides a Spring scope which allows declaring context beans that are stored in the session provided by nibiru. In conjunction with the <aop:scoped-proxy/> tag provided by Spring, this mechanism allows beans which are stored in the session to be transparently injected into singleton beans.

For example:

```
...

<osgi:reference id="nibiruSession"
    interface="ar.com.oxen.nibiru.session.api.Session" />

<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="nibiruSession">
        <bean
            class="ar.com.oxen.nibiru.session.spring.scope.SessionScope">
          <property name="session" ref="nibiruSession" />
        </bean>
      </entry>
    </map>
  </property>
</bean>

<bean name="vaadinApplication" scope="nibiruSession"
    class="ar.com.oxen.nibiru.ui.vaadin.application.NibiruApplication">
  <property name="eventBus" ref="eventBus" />
  <property name="localeHolder" ref="localeHolder" />
  <aop:scoped-proxy />
</bean>

...
```

9 Conversations

A common scenario in business applications includes users operating on a set of data for a given time interval and finally confirming or cancelling pending op-

erations. The conversation (ar.com.oxen.nibiru.conversation.api project) serves as an abstraction of this concept:

```
package ar.com.oxen.nibiru.conversation.api;

/**
 * Interface representing a conversation between the user and the application.
 */
public interface Conversation {
        /**
         * Finishes the conversation OK. Typically, this action is called when a
         * user clicks an "accept" button in order to confirm database changes,
         */
        void end();

        /**
         * Cancels the conversation. Typically called when the user presses a
         * "cancel" button.
         */
        void cancel();

        /**
         * Registers a conversation status tracker.
         *
         * @param tracker
         *            The tracker
         */
        void registerTracker(ConversationTracker tracker);

        /**
         * Activates the conversation and executes the code provided by the
         * callback. Code called from the callback can access the conversation u
         * the {@link ConversationAccessor} service.
         *
         * @param <T>
         *            The type to be returned by the callback
         * @param callback
         *            The callback
         * @return The object returned by the callback
         */
        <T> T execute(ConversationCallback<T> callback);

        /**
         * Gets an object from conversation data.
         *
         * @param <T>
```

```
     *              The object type
     * @param key
     *              The object key (must be unique)
     * @return The object
     */
    <T> T get(String key);

    /**
     * Puts an object into conversation data.
     *
     * @param key
     *              The object key (must be unique)
     * @param value
     *              The object
     */
    void put(String key, Object value);

    /**
     * Removes an object from conversation data.
     *
     * @param key
     *              The object key (must be unique)
     */
    void remove(String key);
}
```

The conversation provides a way to decouple the user interface from the implementation of the various services that require conversation information. For example, suppose you are using the CRUD module with the JPA service implementation. The user interface layer creates a conversation when opening the presenter. With each service call, the CRUD service implementation extracts the active EntityManager from the conversation. Thus, the upper layers doesn't needs to know the details about conversation information needs at lower layers.

To implement this process, the client (usually the presentation layer) creates a conversation using the factory:

```
package ar.com.oxen.nibiru.conversation.api;

/**
 * Conversation factory.
 */
public interface ConversationFactory {
    /**
     * Builds a new conversation.
     *
     * @return The conversation
```

```
         */
        Conversation buildConversation();
}
```

and each time you access a service that requires information from conversation, does it using the execute() method, which receives a callback with a doInConversation() method, which will runs after enabling the conversation:

```
package ar.com.oxen.nibiru.conversation.api;

/**
 * Conversation callback. Used to run code that can access the active
 * conversation using {@link ConversationAccessor}.
 *
 * @param <T>
 */
public interface ConversationCallback<T> {
        /**
         * Method to be executed when conversation is activated.
         *
         * @param conversation
         *            The active conversation
         * @return Anything that the callback would want to return
         * @throws Exception
         *             At any error
         */
        T doInConversation(Conversation conversation) throws Exception;
}
```

Finally, the client can invoke the end() or cancel() methods, in order to either finishing or canceling the conversation.

From lower layers, you can access the active conversation through ConversationAccessor service:

```
package ar.com.oxen.nibiru.conversation.api;

/**
 * Service used to access current active conversation.
 */
public interface ConversationAccessor {
        /**
         * @return The current conversation
         */
        Conversation getCurrentConversation();
}
```

Using get() and put() methods, the component can read and write values from/into the conversation. If you want to perform an action when the conversation terminates/cancels, you can use the registerTracker() to register a callback:

```
package ar.com.oxen.nibiru.conversation.api;

/**
 * Listener for tracking conversation life cycle.
 *
 */
public interface ConversationTracker {
        /**
         * Called when conversation finishes OK.
         *
         * @param conversation
         *              The finished conversation
         */
        void onEnd(Conversation conversation);

        /**
         * Called when conversation is canceled.
         *
         * @param conversation
         *              The canceled conversation
         */
        void onCancel(Conversation conversation);
}
```

The idea of establishing a mechanism comes from Seam conversations, but some modifications were made. First, we aimed to make a simpler design and not being oriented specifically to Web applications. For example, Seam conversations are hierarchical, while those of Nibiru are not. We even hade the idea of unifying the concept of conversation with the session and make it hierachical (being the session the main conversation), but this would add complexity to conversation semantics and force an awkward interface unification, without providing benefits.

The ar.com.oxen.nibiru.conversation.generic module contains generic conversation services implementations.

TODO: Las conversaciones pueden hacer que sea practicamente imposible serializar la sesión.

10 Persistence

JPAis used for persistence. While there are mutliple persistence mechanisms in Java, JPA is the most widespread. For this reason, this specification was chosen over other mechanisms. However, nothing prevents from implementing persistence services using a different technology (of course, this would imply implementing again the modules which depend on JPA).

Since JPA is an API itself, no Nibiru-specific API was defined. On the other hand, an instance of javax.persistence.EntityManagerFactory, from JPA specification, is exposed as a service. The ar.com.oxen.nibiru.jpa.spring bundle provides 2 implementations of such service that use Spring classes:

1. ConversationEntityManagerFactory: Gets the EntityManager form the active conversation, creating it if not exists. Currently this component is exposed as a service.
2. SessionEntityManagerFactory: Gets the EntityManager form the session, creating it if not exists. Is currently evaluating whether this component should be removed (it was the original implementation of the service).

Because JPA requires you to specify in the META-INF/persistence.xml file the classes to be persisted, an OSGi fragment must be created in order to add such file to JPA service bundle. This has the disadvantage that the file should include all the classes to be persisted by different modules. The project ar.com.oxen.nibiru.sample.jpa.fragment is an example of this.

Regarding database access, a javax.sql.DataSource service is exposed . In this case it was not necessary to define a specific Nibiru API. The ar.com.oxen.nibiru.datasource.dbcp bundle provides an implementation using DBCP. The database connection settings and JDBC driver visibility are also added as OSGi fragments. Look at ar.com.oxen.nibiru.sample.datasource.fragment project for an example.

11 User interface

The ar.com.oxen.nibiru.ui.api bundle contains interfaces for presentation layer. The approach aims to build the view using the MVP pattern (passive view). Within the package we have 3 main sub-packages:

1. extension: Contains interfaces to be implemented by UI extensions (currently sub-menu and menu - see Modules section for details).
2. mvp: Contains the interfaces used to implement the MVP pattern: Presenter, View and all necessary ones in order to access to data and events (HasValue, HasClickHandler, clickHandler, etc.). .

3. view: Contains interfaces for view component abstraction. These interfaces are used every time you want to access to a specific widget in a generic way. For example, a button or text field. The idea is to have adapters for the widgets of different UI technologies.

Using this approach, the user has two options for creating a view:

1. In a generic way, ie using an implementation of ar.com.oxen.nibiru.ui.api.view.ViewFactory in order to access generic widget interfaces. This way, a limited user interface can be built, but you can easily change the subjacent technology.
2. Using a specific technology and making the view class implementing the interface used in the MVP. This way you can take advantage of technology characteristics and use graphic editors. In contrast, the changing the technology mean more work.

As the proposed MVP model is passive view, the presenter simply has a reference to an interface that represents the view (at Google the term Display is used). This lets you use either one of the two approaches, without changing the presenter.

In summary, the main MVP interfaces are Presenter:

```
package ar.com.oxen.nibiru.ui.api.mvp;

/**
 * A presenter. The presenter should contain the presentation logic, in order to
 * keep it decoupled from the view.
 *
 * @param <V>
 *            The view type.
 */
public interface Presenter<V extends View> {
        /**
         * Activates the presenter. This method is called after setting the view
         * Typically, this method will add listeners for presentation logic that
         * reacts to view events.
         */
        void go();

        /**
         * @param view
         *            The view to be used with the presenter
         */
        void setView(V view);
}
```

and View:

```
package ar.com.oxen.nibiru.ui.api.mvp;

/**
 * A view. Implementations of this interface shouldn't contain presentation
 * logic. Instead, display-related logic, such as layout setup, text
 * internationalization, etc should be responsibility of View implementations.
 *
 */
public interface View {
        /**
         * Shows the view.
         */
        void show();

        /**
         * Closes the view.
         */
        void close();
}
```

The presentation logic should be put on the method go() of Presenter class.

Widgets abstraction interfaces (ar.com.oxen.nibiru.ui.api.view package) are varied. But all should be instantiated by an implementation of ViewFactory:

```
package ar.com.oxen.nibiru.ui.api.view;

/**
 * Builds components (widgets, windows, etc) to be used in views. The purpose of
 * this interface is hiding UI framework specific implementations.
 */
public interface ViewFactory {
        /**
         * Builds a main window.
         *
         * @return The main window.
         */
        MainWindow buildMainWindow();

        /**
         * Builds a window.
         *
         * @return The window
         */
        Window buildWindow();

        /**
```

```
     * Builds a label.
     *
     * @param <T>
     *            The type of data to be shown by the label. Typically Strin
     * @param type
     *            The class of data to be shown by the label. Typically Stri
     * @return The label
     */
    <T> Label<T> buildLabel(Class<T> type);

    /**
     * Builds a button.
     *
     * @return The button.
     */
    Button buildButton();

    /**
     * Builds a text field.
     *
     * @param <T>
     *            The type of data to be shown by the text field. Typically
     *            String.
     * @param type
     *            The class of data to be shown by the text field. Typically
     *            String.
     * @return The text field
     */
    <T> TextField<T> buildTextField(Class<T> type);

    /**
     * Builds a password field.
     *
     * @param <T>
     *            The type of data to be shown by the password field. Typica
     *            String.
     * @param type
     *            The class of data to be shown by the password field. Typic
     *            String.
     * @return The password field
     */
    <T> PasswordField<T> buildPasswordField(Class<T> type);

    /**
     * Builds a multiline text area.
     *
```

```
     * @param <T>
     *             The type of data to be shown by the password field. Typica
     *             String.
     * @param type
     *             The class of data to be shown by the password field. Typic
     *             String.
     * @return The text area
     */
    <T> TextArea<T> buildTextArea(Class<T> type);

    /**
     * Builds a date field.
     *
     * @return The date field
     */
    DateField buildDateField();

    /**
     * Builds a time field.
     *
     * @return The time field
     */
    TimeField buildTimeField();

    /**
     * Builds a check box.
     *
     * @return The check box
     */
    CheckBox buildCheckBox();

    /**
     * Builds a combo box.
     *
     * @param <T>
     *             The type of data to be shown by the combo.
     * @param type
     *             The class of data to be shown by the combo.
     * @return The combo box
     */
    <T> ComboBox<T> buildComboBox(Class<T> type);

    /**
     * Builds a list select.
     *
     * @param <T>
```

```
     *            The type of data to be shown by the list select.
     * @param type
     *            The class of data to be shown by the list select.
     * @return The list select
     */
    <T> ListSelect<T> buildListSelect(Class<T> type);

    /**
     * Builds a table.
     *
     * @return The table
     */
    Table buildTable();

    /**
     * Builds a panel with vertical layout.
     *
     * @return The panel.
     */
    Panel buildVerticalPanel();

    /**
     * Builds a panel with horizontal layout.
     *
     * @return The panel.
     */
    Panel buildHorizontalPanel();

    /**
     * Builds a panel with form layout.
     *
     * @return The panel.
     */
    Panel buildFormPanel();

    /**
     * Builds a tabbed panel.
     *
     * @return The panel
     */
    Panel buildTabPanel();
}
```

The ar.com.oxen.nibiru.ui.vaadin project contains a factory and its associated adapters required in order to implement ar.com.oxen.nibiru.ui.api.view interfaces using Vaadin.

The ar.com.oxen.nibiru.ui.utils project contains generic classes for use in the user interface. Mostly contains abstract classes to be used as base for presenters , views, extensions, etc. But also contains decorators and generic use classes.

12 Security

The interfaces required for accessing security services (authentication and authorization) are found in the ar.com.oxen.nibiru.security.api project . Currently user/password authentication and key role authorization are supported.

Authentication is done through the AuthenticationService interface:

```
package ar.com.oxen.nibiru.security.api;

/**
 * Service for authenticating users.
 */
public interface AuthenticationService {
        /**
         * Performs an user log-on.
         *
         * @param user
         *             The user name
         * @param password
         *             The password
         * @throws BadCredentialsException
         *              If the user name and/or the password is not valid
         */
        void login(String user, String password) throws BadCredentialsException;

        /**
         * Performs an user log-off.
         */
        void logout();

        /**
         * @return The login name of the logged user (if any).
         */
        String getLoggedUserName();
}
```

While authorization is performed by AuthorizationService:

```
package ar.com.oxen.nibiru.security.api;

/**
```

```
 * Service for authorizing actions and users.
 *
 */
public interface AuthorizationService {
        /**
         * Checks if the logged user has a given role.
         *
         * @param role
         *             The role name.
         * @return True if the user has the role
         */
        boolean isCallerInRole(String role);
}
```

TODO: Habría que pensar bien el esquema de autorización. Si los módulos se van a desarrollar de forma independiente, cómo se evita la colisión de roles? Depende de la implementación asociar los roles específicos de cada módulo a un rol general? (como en EJB) O simplemente que cada modulo le ponga un prefijo al rol? (como se está haciendo con la internacionalizacion).

So far there has been no security implementation, just one dummy service provided by ar.com.oxen.nibiru.security.dummy bundle.

13 Transaction management

Since there are not intrusive mechanisms (using AOP), no specific API was defined in this case. It would be defined in order to provide programmatic transaction management.

The ar.com.oxen.nibiru.transaction.spring bundle exposes a Spring TransactionManager as an OSGi service. Within each bundle Spring AOP may be used in order to, declaratively, setting transactions (by injecting the TransactionManager service).

For example:

```
...

<osgi:reference id="transactionManager"
    interface="org.springframework.transaction.PlatformTransactionManager" />

<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

```
<bean   class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostP

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreato
  <property name="beanNames" value="dynamicBundleCrudManager"/>
  <property name="interceptorNames" value="txAdvice"/>
</bean>

...
```

Currently, not all the Spring XML tags for transaction management are supported, so you should use a BeanNameAutoProxyCreator component, as shown in the example.

TODO: El bundle expone un JpaTransactionManager. El nombre del proyecto debería decir "jpa" en algún lugar.

14 Internationalization

The ar.com.oxen.nibiru.i18n.api project contains interfaces for internationalization. There are 3 main services:

1. LocaleHolder: Used to read or write the user's Locale.
2. MessageSource: Used to get messages by key (with parameters).
3. MessageProvider: Used to provide message querying using a key and a Locale. This division was made so that each module can provide its own MessageProvider. Typically there will be a MessageSource implementation that consolidates them.

The 3 interfaces are very simple, as you can see.

LocaleHolder:

```
package ar.com.oxen.nibiru.i18n.api;

import java.util.Locale;

/**
 * Service used to access the user locale.
 *
 */
public interface LocaleHolder {
        /**
         * Gets the user locale.
```

```
         *
         * @return The locale
         */
        Locale getLocale();

        /**
         * Sets the user locale.
         *
         * @param newLocale
         *            The locale
         */
        void setLocale(Locale newLocale);
}
```

MessageSource:

```
package ar.com.oxen.nibiru.i18n.api;

import java.util.Locale;

/**
 * Service for accessing i18n messages. Typically a view from a module will
 * access this service. Internally, implementation of this module should access
 * the current user locale with {@link LocaleHolder} and delegate on N
 * {@link MessageProvider}s in order to look for the searched message.
 */
public interface MessageSource {
        /**
         * Gets a i18n message
         *
         * @param code
         *            The message code
         * @param args
         *            The message arguments
         * @return The translated an parsed message. If the message is not found
         *         returns null.
         */
        String getMessage(String code, Object... args);

        /**
         * Returns a 18n message
         *
         * @param code
         *            The message code
         * @param locale
         *            The locale
         * @param args
```

```
         *              The message arguments
         * @return The translated an parsed message. If the message is not found
         *          null is returned.
         */
        String getMessage(String code, Locale locale, Object... args);
}
```

MessageProvider:

```
package ar.com.oxen.nibiru.i18n.api;

import java.util.Locale;

/**
 * A message provider. This interface is provided in order to allow i18n
 * modularity. Each module could provide its own MessageProvider. All the
 * MessageProviders would be consolidated by a single, generic
 * {@link MessageSource}.
 */
public interface MessageProvider {
        /**
         * Returns a 18n message
         *
         * @param code
         *              The message code
         * @param locale
         *              The locale
         * @param args
         *              The message arguments
         * @return The translated an parsed message. If the message is not found
         *          null is returned.
         */
        String getMessage(String code, Locale locale, Object... args);
}
```

The ar.com.oxen.nibiru.i18n.generic project contains an generic MessageSource implementation which is injected with LocaleHolder and a list of MessageProviders. Spring DM can inject a MessageProvider service list that is updated dynamically according to the availability of new instances of these services. This project also contains a MessageProvider implementation based on ResoruceBundle.

The ar.com.oxen.nibiru.i18n.session project has a LocaleHolder implementation that stores the locale in the Nibiru session.

15 CRUD

CRUD module (Create, Read, Update and Delete) aims to facilitate the generation of funcionality of this type.

The functionality of this module is distributed across multiple bundles. It can be grouped into 2 layers.

TODO: pensar cómo integrar seguridad al generador de ABMS.

15.1 Persistence services

The required interfaces for exposing persistence services are found in the ar.com.oxen.nibiru.crud.manager.api project.

The main interface is CrudManager, which provides the necessary methods to dynamically generate an CRUD screen. In other words, the idea is to have a CrudManager by each entity on which you want to build a CRUD.

```
package ar.com.oxen.nibiru.crud.manager.api;

import java.util.List;

/**
 * Service for managing CRUD over entities.
 *
 * @param <T>
 *            The crud entity type.
 */
public interface CrudManager<T> {
        /**
         * Returns the entity type name.
         *
         * The name identifies the kind of entity being handled. This is useful,
         * example, in order to determine if a given entity is compatible with a
         * crud manager.
         *
         * @return The type name.
         */
        String getEntityTypeName();

        /**
         * Gets the fields to be shown in the entity list.
         *
         * @return A list with the fields
         */
        List<CrudField> getListFields();
```

```
        /**
         * Gets the fields to be shown in the entity form.
         *
         * @return A list with the fields
         */
        List<CrudField> getFormFields();

        /**
         * Reads all the entities.
         *
         * @return A list with the entities
         */
        List<CrudEntity<T>> findAll();

        /**
         * Reads entities filtering by a given field. Useful for parent-child
         * relations
         *
         * @return A list with the entities
         */
        List<CrudEntity<T>> findByfield(String field, Object value);

}
```

CRUD module is designed for handling various types of entities. Unlike a typical CRUD generator, where screens are generated to manage tables in a database or on beans, Nibiru CRUD adds a level of indirection. This allows you to create persistence service implementations providing access to beans JPA, business process instances, and so on.

The interfaces used to achieve this level of abstraction are CrudEntity (representing an entity that is being edited) and CrudField (which represents a field of such entity).

```
package ar.com.oxen.nibiru.crud.manager.api;

/**
 * Represents an entity instance. This interface is used in order to hide entity
 * implementation. This way, CRUD engine could work over Java beans, BPM
 * processes, etc.
 *
 * @param <T>
 */
public interface CrudEntity<T> {
        /**
         * Reads a field value.
```

```
     *
     * @param field
     *            The field
     * @return The value
     */
    Object getValue(CrudField field);

    /**
     * Reads a field value.
     *
     * @param fieldName
     *            The field name
     * @return The value
     */
    Object getValue(String fieldName);

    /**
     * Writes a field value
     *
     * @param field
     *            The field
     * @param value
     *            The value
     */
    void setValue(CrudField field, Object value);

    /**
     * Writes a field value
     *
     * @param fieldName
     *            The field name
     * @param value
     *            The value
     */
    void setValue(String fieldName, Object value);

    /**
     * Gets the wrapped object.
     *
     * @return The entity object
     */
    T getEntity();

    /**
     * Returns the entity type name.
     *
```

```
         * The name identifies the kind of entity being handled. This is useful,
         * example, in order to determine if a given entity is compatible with a
         * crud manager.
         *
         * @return The type name.
         */
        String getEntityTypeName();

        /**
         * Returns the available values for a given field (for example, for usin
         * a combo box or a list select)
         *
         * @param field
         *            The field
         * @return An iterable for the values
         */
        Iterable<Object> getAvailableValues(CrudField field);

        /**
         * Returns the available values for a given field (for example, for usin
         * a combo box or a list select)
         *
         * @param fieldName
         *            The field name
         * @return An iterable for the values
         */
        Iterable<Object> getAvailableValues(String fieldName);
}

package ar.com.oxen.nibiru.crud.manager.api;

/**
 * Represents a field on a {@link CrudEntity}.
 *
 */
public interface CrudField {
        /**
         * @return The field name
         */
        String getName();

        /**
         * @return The field class
         */
        Class<?> getType();
```

```
/**
 * @return Information for showing the field in a list.
 */
ListInfo getListInfo();

/**
 * @return Information for showing the field in a form.
 */
FormInfo getFormInfo();

/**
 * Information for showing the field in a list.
 */
interface ListInfo {
        /**
         * Determines a fixed width for the field column.
         *
         * @return The column width
         */
        int getColumnWidth();

}

/**
 * Information for showing the field in a form.
 */
interface FormInfo {
        /**
         * Determines how the field should be represented (for example,
         * form).
         *
         * @return An element of widget type enumeration
         */
        WidgetType getWidgetType();

        /**
         * @return True if the field can't be modified
         */
        boolean isReadonly();

        /**
         * Determines how many characters can be set on the field. Appli
         * to widgets which holds Strings.
         *
         * @return The maximum length
         */
```

```
                int getMaxLength();

                /**
                 * Returns the tab name where the widget must be shown.
                 *
                 * @return The tab name
                 */
                String getTab();
        }
}
```

WidgetType enumerates the ways in which a field can be shown:

```
package ar.com.oxen.nibiru.crud.manager.api;

public enum WidgetType {
        TEXT_FIELD,
        PASSWORD_FIELD,
        TEXT_AREA,
        DATE_FIELD,
        TIME_FIELD,
        CHECK_BOX,
        COMBO_BOX,
        MULTISELECT
}
```

The abstraction would not be complete if the actions to be performed on the entities weren't not configurable. To this end the CrudAction interface was created.

```
package ar.com.oxen.nibiru.crud.manager.api;

/**
 * Represents an action that can be applied on a CRUD. Abstracting the actions
 * allows the CRUD implementations to provide extra actions. This way, actions
 * are not limited to create, read, update an delete (so the module shouldn't be
 * called CRUD!!!), but can add action such as approve, reject, start, stop,
 * etc. In some cases, the action can require no entity (for example, "new"). In
 * other cases, it would be mandatory applying the action over an specific
 * {@link CrudEntity} ("edit", for example).
 *
 */
public interface CrudAction {
        String NEW = "new";
        String DELETE = "delete";
        String EDIT = "edit";
        String UPDATE = "update";
```

```
        /**
         * Gets the action name.
         *
         * @return The name
         */
        String getName();

        /**
         * Indicates if the action must be performed over an {@link CrudEntity}.
         *
         * @return True if a {@link CrudEntity} is required
         */
        boolean isEntityRequired();

        /**
         * Indicates if a user confirmation must be presented before performing
         * action.
         *
         * @return True if confirmation must be presented
         */
        boolean isConfirmationRequired();

        /**
         * Indicates if the action must be shown in list window.
         *
         * @return True if it must be shown
         */
        boolean isVisibleInList();

        /**
         * Indicates if the action must be shown in form window.
         *
         * @return True if it must be shown
         */
        boolean isVisibleInForm();
}
```

In this way the actions are not limited to create, read, update and delete, but they are extensible. A workflow engine could, for example, display actions such as "approve" or "reject."

In order to make the CRUD modular, the actions to perform on an entity are not provided directly by the CrudManager, but using the extension point mechanism. The interface CrudActionExtension allows implementing extensions that add different possible actions to perform on an entity.

```
package ar.com.oxen.nibiru.crud.manager.api;

import java.util.List;

/**
 * Extension used to add actions to CRUD.
 *
 * @param <T>
 *            The {@link CrudEntity} type
 */
public interface CrudActionExtension<T> {
        /**
         * Get actions provided by this extension.
         *
         * @return A list with the actions
         */
        List<CrudAction> getActions();

        /**
         * Performs an action over a given entity. The action can create/update
         * entity. In that case, such entity is returned, otherwise it returns n
         * When a created/updated entity is returned, the CRUD should open a form
         * order to edit it. This can be useful, for example, for BPM
         * implementations that jumps from an activity to another.
         *
         * @param action
         *            The action
         * @param entity
         *            The entity (it can be null if the action doesn't require a
         *            entity)
         * @return The created/updated entity
         */
        CrudEntity<T> performAction(CrudAction action, CrudEntity<T> entity);
}
```

The ar.com.oxen.nibiru.crud.manager.jpa bundle contains implementations based on JPA. It relies on ar.com.oxen.nibiru.crud.bean and ar.com.oxen.nibiru.crud.utils classes. Where possible, it uses JPA information and reflection to return the information required for CRUD. Where not possible, it uses ar.com.oxen.nibiru.crud.bean based on annotations.

15.1.1 Events

The CRUD API provides some common use events. They are intended to be used when communicating the different CRUD components through the event bus.

The ManageCrudEntitiesEvent can be used in order to notify that administration of entities of a given type is required. This event is tipically fired from a menu.

```
package ar.com.oxen.nibiru.crud.manager.api;

/**
 * This is a generic event class for triggering entities management. The topic
 * should be used in order to identify the entity to be managed.
 */
public class ManageCrudEntitiesEvent {
}
```

The EditCrudEntityEvent indicates that a given entity must be edited This tipically will open a CRUD form.

```
package ar.com.oxen.nibiru.crud.manager.api;

import ar.com.oxen.nibiru.conversation.api.Conversation;

public class EditCrudEntityEvent {
        private CrudEntity<?> entity;
        private Conversation conversation;

        public EditCrudEntityEvent(CrudEntity<?> entity, Conversation conversati
                super();
                this.entity = entity;
                this.conversation = conversation;
        }

        public CrudEntity<?> getCrudEntity() {
                return entity;
        }

        public Conversation getConversation() {
                return conversation;
        }
}
```

When editing is finished, a ModifiedCrudEntityEvent can be fired in order to notify that such instance has been modified. For example, the CRUD list presenter listens to this event in order to refresh the list.

```
package ar.com.oxen.nibiru.crud.manager.api;

public class ModifiedCrudEntityEvent {
        private CrudEntity<?> entity;
```

```
        public ModifiedCrudEntityEvent(CrudEntity<?> entity) {
                super();
                this.entity = entity;
        }

        public CrudEntity<?> getCrudEntity() {
                return entity;
        }
}
```

Finally, a ManageChildCrudEntitiesEvent can be fired in order to activate a CRUD for dependant entities (in a parent-child relationship).

```
package ar.com.oxen.nibiru.crud.manager.api;

/**
 * This is a generic event class for triggering entities management related to a
 * parent. The topic should be used in order to identify the entity to be
 * managed.
 */
public class ManageChildCrudEntitiesEvent {
        private String parentField;
        private Object parentEntity;

        public ManageChildCrudEntitiesEvent(String parentField, Object parentEnt
                super();
                this.parentField = parentField;
                this.parentEntity = parentEntity;
        }

        public String getParentField() {
                return parentField;
        }

        public Object getParentEntity() {
                return parentEntity;
        }
}
```

15.2 User interface services

The ar.com.oxen.nibiru.crud.ui.api project contains interfaces for CRUD views and presenters.

These interfaces must be instantiated by a presenter factory implementation:

```java
package ar.com.oxen.nibiru.crud.ui.api;

import ar.com.oxen.nibiru.crud.manager.api.CrudManager;
import ar.com.oxen.nibiru.crud.manager.api.EditCrudEntityEvent;
import ar.com.oxen.nibiru.crud.ui.api.form.CrudFormView;
import ar.com.oxen.nibiru.crud.ui.api.list.CrudListView;
import ar.com.oxen.nibiru.ui.api.mvp.Presenter;

/**
 * CRUD presenter factory.
 */
public interface CrudPresenterFactory {
        /**
         * Builds a presenter for CRUD list.
         *
         * @param crudManager
         *              The CRUD manager
         * @return The presenter
         */
        Presenter<CrudListView> buildListPresenter(CrudManager<?> crudManager);

        /**
         * Builds a presenter for CRUD list which is filtered by a parent value.
         *
         * @param crudManager
         *              The CRUD manager
         * @param parentField
         *              The field used in order to filter the parent value.
         * @param parentValue
         *              The parent value.
         * @return The presenter
         */
        Presenter<CrudListView> buildListPresenter(CrudManager<?> crudManager,
                        String parentField, Object parentValue);

        /**
         * Builds a presenter for CRUD form.
         *
         * @param crudManager
         *              The CRUD manager
         * @return The presenter
         */
        Presenter<CrudFormView> buildFormPresenter(CrudManager<?> crudManager,
                        EditCrudEntityEvent event);
}
```

and a view factory:

```
package ar.com.oxen.nibiru.crud.ui.api;

import ar.com.oxen.nibiru.crud.ui.api.form.CrudFormView;
import ar.com.oxen.nibiru.crud.ui.api.list.CrudListView;

/**
 * CRUD presenter factory.
 */
public interface CrudViewFactory {
        String I18N_FIELD_PREFIX = "ar.com.oxen.nibiru.crud.field.";
        String I18N_ACTION_PREFIX = "ar.com.oxen.nibiru.crud.action.";
        String I18N_ENTITY_PREFIX = "ar.com.oxen.nibiru.crud.entity.";
        String I18N_TAB_PREFIX = "ar.com.oxen.nibiru.crud.tab.";
        String I18N_ERROR_PREFIX = "ar.com.oxen.nibiru.crud.error.";

        /**
         * Builds the view for CRUD list.
         *
         * @return The view
         */
        CrudListView buildListView();

        /**
         * Builds the view for CRUD form.
         *
         * @return The view
         */
        CrudFormView buildFormView();
}
```

There is a generic implementation in the ar.com.oxen.nibiru.crud.ui.generic project.

15.3 Utilities

The ar.com.oxen.nibiru.crud.utils bundle contains generic utility classes for creating CRUDs. This includes:

- Simple implementations for CrudField and CrudAction.
- Common action extensions.
- A base class for CRUD modules configuration (AbstractCrudModuleConfigurator).

The AbstractCrudModuleConfigurator class provides the following methods:

- addCrud: Adds a top-level CRUD, which are started from application menu. The method registers the extension points for menu and actions. Also, it registers event bus listeners for navigation.
- addChildCrud: Adds a child CRUD, which is fired from a parent CRUD contextual menu. In a similar way, it registers the appropiate extensions and listeners.

```
package ar.com.oxen.nibiru.crud.utils;

import java.util.LinkedList;
import java.util.List;

import ar.com.oxen.commons.eventbus.api.EventHandler;
import ar.com.oxen.nibiru.crud.manager.api.CrudActionExtension;
import ar.com.oxen.nibiru.crud.manager.api.CrudManager;
import ar.com.oxen.nibiru.crud.manager.api.EditCrudEntityEvent;
import ar.com.oxen.nibiru.crud.manager.api.ManageChildCrudEntitiesEvent;
import ar.com.oxen.nibiru.crud.manager.api.ManageCrudEntitiesEvent;
import ar.com.oxen.nibiru.crud.ui.api.CrudPresenterFactory;
import ar.com.oxen.nibiru.crud.ui.api.CrudViewFactory;
import ar.com.oxen.nibiru.module.utils.AbstractModuleConfigurator;
import ar.com.oxen.nibiru.ui.api.extension.MenuItemExtension;
import ar.com.oxen.nibiru.ui.utils.extension.SimpleMenuItemExtension;
import ar.com.oxen.nibiru.ui.utils.mvp.SimpleEventBusClickHandler;

public abstract class AbstractCrudModuleConfigurator extends
                AbstractModuleConfigurator<CrudViewFactory, CrudPresenterFactory
        private List<EventHandler<?>> registeredHandlers = new LinkedList<EventH

        int menuPos = 0;

        /**
         * Adds a CRUD menu
         */
        protected <K> void addCrudMenu(String menuName, String parentMenuExtensi
                        CrudManager<K> crudManager) {
                this.registerMenu(menuName, parentMenuExtension, crudManager);
        }

        /**
         * Adds a CRUD without a menu option
         */
        protected <K> void addCrud(CrudManager<K> crudManager,
                        CrudActionExtension<K> crudActionExtension) {
                this.registerManageEntityEvent(crudManager);
```

```
        this.registerActions(crudManager, crudActionExtension);
        this.registerEditEntityEvent(crudManager);
}

/**
 * Adds a CRUD with a menu option
 */
protected <K> void addCrudWithMenu(String menuName,
                String parentMenuExtension, CrudManager<K> crudManager,
                CrudActionExtension<K> crudActionExtension) {
        this.addCrudMenu(menuName, parentMenuExtension, crudManager);
        this.addCrud(crudManager, crudActionExtension);
}

/**
 * Adds a child menu CRUD menu option
 */
protected <T> void addChildCrudMenu(String menuName,
                CrudManager<?> parentCrudManager, String parentField,
                CrudManager<T> childCrudManager) {

        this.registerManageChildrenAction(menuName, parentCrudManager,
                        childCrudManager, parentField);
}

/**
 * Adds a child menu CRUD without a menu option
 */
protected <T> void addChildCrud(CrudManager<?> parentCrudManager,
                CrudManager<T> childCrudManager,
                CrudActionExtension<T> childCrudActionExtension) {

        this.registerActions(childCrudManager, childCrudActionExtension)
        this.registerManageChildEntitiesEvent(parentCrudManager,
                        childCrudManager);
        this.registerEditEntityEvent(childCrudManager);
}

/**
 * Adds a child menu CRUD with a menu option
 */
protected <T> void addChildCrudWithMenu(String menuName,
                CrudManager<?> parentCrudManager, String parentField,
                CrudManager<T> childCrudManager,
                CrudActionExtension<T> childCrudActionExtension) {
```

```
        this.addChildCrudMenu(menuName, parentCrudManager, parentField,
                childCrudManager);
        this.addChildCrud(parentCrudManager, childCrudManager,
                childCrudActionExtension);
}

@Override
public void shutdown() {
        super.shutdown();
        for (EventHandler<?> handler : this.registeredHandlers) {
                this.getEventBus().removeHandler(handler);
        }
}

private void registerMenu(String menuName, String parentMenuExtension,
                CrudManager<?> crudManager) {
        this.registerExtension(new SimpleMenuItemExtension(menuName, men
                        new SimpleEventBusClickHandler(this.getEventBus(
                                        ManageCrudEntitiesEvent.class, c
                                                        .getEntityTypeNa
                        MenuItemExtension.class);
}

private <K> void registerActions(CrudManager<K> crudManager,
                CrudActionExtension<K> crudActionExtension) {
        this.registerExtension(crudActionExtension, crudManager
                        .getEntityTypeName(), CrudActionExtension.class)
}

private <K> void registerManageChildrenAction(String menuName,
                CrudManager<?> parentCrudManager,
                final CrudManager<?> childCrudManager, String parentFiel
        this.registerExtension(new ManageChildrenCrudActionExtension<Obj
                        menuName, parentField, childCrudManager.getEntity
                        this.getEventBus()), parentCrudManager.getEntityT
                        CrudActionExtension.class);
}

private void registerManageEntityEvent(final CrudManager<?> crudManager)
        this.addEventHandler(ManageCrudEntitiesEvent.class,
                        new EventHandler<ManageCrudEntitiesEvent>() {

                                @Override
                                public void onEvent(ManageCrudEntitiesEv
                                        activate(getViewFactory().buildL
                                                        getPresenterFact
```

```
                        }
                    }, crudManager.getEntityTypeName());
    }

    private void registerEditEntityEvent(final CrudManager<?> crudManager) {
        this.addEventHandler(EditCrudEntityEvent.class,
                new EventHandler<EditCrudEntityEvent>() {

                    @Override
                    public void onEvent(EditCrudEntityEvent
                        activate(getViewFactory().buildFo
                                getPresenterFact

                    }
                }, crudManager.getEntityTypeName());
    }

    private void registerManageChildEntitiesEvent(
            CrudManager<?> parentCrudManager,
            final CrudManager<?> childCrudManager) {
        this.addEventHandler(ManageChildCrudEntitiesEvent.class,
                new EventHandler<ManageChildCrudEntitiesEvent>()

                    @Override
                    public void onEvent(ManageChildCrudEntit
                        activate(getViewFactory().buildL
                                getPresenterFact



                    }
                }, childCrudManager.getEntityTypeName());
    }

    private <T> void addEventHandler(Class<T> eventClass,
            EventHandler<T> handler, String topic) {
        this.registeredHandlers.add(handler);
        this.getEventBus().addHandler(eventClass, handler, topic);
    }
}
```

The ar.com.oxen.nibiru.crud.bean project contains utility classes for CRUD implementations that use beans, like an implementation of CrudEntity that delegates to a bean (through BeanWrapper of Java Oxen Commons). Also, it contains annotations which are useful in order to to parametrize the CRUD

directly on the bean.

16 Reports

TODO: Definir este módulo.

17 Workflow

TODO: Definir este módulo.

18 Dynamic bundles

The goal of dynamic bundles mechanism is to provide, in conjunction with the extension point mechanism, a way to add functionality or customize the system while it is running.

OSGi already provides a service to install bundles dynamically. The idea of the module provided by Nibiru is to provide a mechanism for such bundles to be stored in the database and managed from the same application. It is worth noting that the service of Nibiru is not generic, but is designed specifically for OSGi.

In conjunction with various scripting engines, you can add functionality without recompiling any bundle.

18.1 Persistence

The ar.com.oxen.nibiru.dynamicbundle.domain bundle contains domain classes used to persist bundles. The ar.com.oxen.nibiru.dynamicbundle.dao.api bundle provides a DAO used to read and write these domain classes.

```
package ar.com.oxen.nibiru.dynamicbundle.dao.api;

import ar.com.oxen.commons.dao.api.ReadDao;
import ar.com.oxen.commons.dao.api.UpdateDao;
import ar.com.oxen.nibiru.dynamicbundle.domain.DynamicBundle;

/**
 * DAO for accessing dynamic bundles.
 */
public interface DynamicBundleDao extends ReadDao<DynamicBundle>,
                    UpdateDao<DynamicBundle> {
}
```

Currently there is a JPA implementation of the DAO: ar.com.oxen.nibiru.dynamicbundle.dao.jpa.

TODO: definir mejor el modelo de datos de bundles dinámicos. Por ejemplo, sólo tiene dependencias por bundle, cuando en realidad es mejor poner dependencias por paquete. O también se podría pensar cómo hacer para que además de tener archivos de Spring pueda tener código Java compilado (binario).

TODO: La persistencia no se podría hacer por el manager del módulo de ABMs? Actualmente, para lo único que se está usando el DynamicBundleDao, desde SpringDynamicBundleManager, es para leer todos los bundles e inicializarlos al arrancar.

18.2 Business

The ar.com.oxen.nibiru.dynamicbundle.manager.api project provides the DynamicBundleManager interface, which can be used to operate on bundles.

```
package ar.com.oxen.nibiru.dynamicbundle.manager.api;

import ar.com.oxen.nibiru.dynamicbundle.domain.DynamicBundle;

/**
 * Service for managing dynamic bundles.
 */
public interface DynamicBundleManager {
        /**
         * Deploys and stars a dynamic bundle.
         *
         * @param dynamicBundle
         *              The dynamic bundle
         */
        void start(DynamicBundle dynamicBundle);

        /**
         * Stops and undeploys a dynamic bundle.
         *
         * @param dynamicBundle
         *              The dynamic bundle
         */
        void stop(DynamicBundle dynamicBundle);
}
```

In ar.com.oxen.nibiru.dynamicbundle.manager.spring there is an implementation that uses Spring DM in order to accessing the BundleContext.

To start and stop services, the action extension mechanism provided by the CRUD module is used. The DynamicBundleStatusExtension class from ar.com.oxen.nibiru.dynamicbundle.mod

bundle provides CrudActions to start and stop a given service (delegating on DynamicBundleManager).

```
package ar.com.oxen.nibiru.dynamicbundle.module;

import java.util.ArrayList;
import java.util.List;

import ar.com.oxen.nibiru.crud.utils.SimpleCrudAction;
import ar.com.oxen.nibiru.crud.manager.api.CrudAction;
import ar.com.oxen.nibiru.crud.manager.api.CrudActionExtension;
import ar.com.oxen.nibiru.crud.manager.api.CrudEntity;
import ar.com.oxen.nibiru.dynamicbundle.domain.DynamicBundle;
import ar.com.oxen.nibiru.dynamicbundle.manager.api.DynamicBundleManager;

/**
 * CRUD action extension for starting and stopping dynamic bundles. It delegates
 * on {@link DynamicBundleManager}.
 *
 */
public class DynamicBundleStatusExtension implements
                CrudActionExtension<DynamicBundle> {
        private List<CrudAction> actions;
        private final static String START = "start";
        private final static String STOP = "stop";
        private DynamicBundleManager dynamicBundleManager;

        public DynamicBundleStatusExtension() {
                super();
                this.actions = new ArrayList<CrudAction>(2);
                this.actions.add(new SimpleCrudAction(START, true, false, true,
                this.actions.add(new SimpleCrudAction(STOP, true, false, true, f
        }

        @Override
        public List<CrudAction> getActions() {
                return this.actions;
        }

        @Override
        public CrudEntity<DynamicBundle> performAction(CrudAction action,
                        CrudEntity<DynamicBundle> entity) {
                if (START.equals(action.getName())) {
                        this.dynamicBundleManager.start(entity.getEntity());
                        return null;
                } else if (STOP.equals(action.getName())) {
```

```
                        this.dynamicBundleManager.stop(entity.getEntity());
                        return null;
                } else {
                        throw new IllegalArgumentException("Invalid action: " +
                }
        }

        public void setDynamicBundleManager(
                        DynamicBundleManager dynamicBundleManager) {
                this.dynamicBundleManager = dynamicBundleManager;
        }
}
```

18.3 User interface

At the ar.com.oxen.nibiru.dynamicbundle.module bundle the events that trigger the activation of various presenters and views are setup. There is no specific project for UI because it is based on services provided by the CRUD UI module.

19 Log

TODO: Es necesario un servicio de log? o simplemente con commons loggin o SLF4J alcanza? OSGi tiene un servicio de log, se podría hacer un adaptador para SLF4J. Y seguir la misma lógic que con transacciones, JPA y DataSource.

Part III

License

The framework is distributed under Apache 2.0 license.