Hippikon
Open Source Authorization API for Java Applications

Documentation & Developer Reference Guide

Introduction

The Hippikon Authorization framework provides an object-oriented, white-box framework that allows for complex authorization security models to be configured through XML, and enhanced by a callback mechanism if ruther runtime checking is required. The system is based on sets of permissions created by taking user associations, object state/behaviour and object hierachy into account.

The following list provides an overview of Hippikon's features:

- Authorization logic and permission ACLs is decopuled from business objects
- Policy rules may be reused across different products in an application domain
- Permissions for each product are stored in a persistent PolicyStore that
  - Supports n-deep nested objects
  - Supports different permissions for objects that change with regard to their object hierachy
  - Supports inheritance and overrides of permissions within a object hierachy
- Clients obtain permissions from the PermissionsFactory class by way of a single API call
- Integrates easily into any Java application and is agnostic of frameworks such as J2EE, Struts or Spring.
- The API is based on standard security Design Patterns

Who Should Read This Document

This document is intended for developers who create new products written in Java or who work on authorization logic within Java applications.

Related Documentation

This document assumes you are already familiar with Java 1.2 Standard Edition development. Prior security domain knowledge is preferred but not essential. You should also be familiar with XML, HTTP and some basic encryption terminology.

Other sources of information that influenced the API include:

- JAAS White Paper
- Java Cryptography Extension
- Patterns for Enabling Application Security - Yoder & Barcalow [a must read]
- Martin Fowler: Dealing with Roles
- Practical UNIX Security
- DCE Overview
- LDAP Security

Terminology

ACL	*Access Control List*
Principal	*an association between a user and a system or object an ACL can be defined for*
Role	*a type of principal that cuts across all users and all objects*
Credentials	*a piece of data that identifies a user*
Authentication	*the mechanism of verifying a user based on supplied credentials*
Authorization	*the mechansim of determining what actions an authenticated user can perform*
Policy	*one or more rules that determine authorization rules for a user*
Product	*a specific application that provides services to a user*
Account	*an entity or organization that contains users*
Product Subscription	*a product that an account and its users has been granted access to*

Core Classes and Interfaces

The Hippikon core classes, interfaces and mechanisms can be broken into 3 categories: Common, Authorization and SDK Reference.

- Common Classes
  - AuthorizationContext, PermissionsFactory, ProtectedResource, IllegalAuthorizationException
- Authorization Classes
  - Policy, PolicyStore,
- SDK Reference
  - Creating Policy implementations, Implementing specialized Policy classes, The resource.policies file, Policy Store Extensions Cache

Common Classes

Common classes are those classes with which the application logic interacts to determine the permissions of a user at runtime. All other classes are internal to the SSF or part of the SDK.

com.hippikon.security.AuthorizationContext

A key Hippikon class is `com.hippikon.security.AuthorizationContext`, which represents a user who has been authenticated by the an applications Authentication mechanism or subsystem and the product they attempt to access. It encompasses the entity's role principals, user identifier, the customer or user account to which they belong, the products to which their customer or user account has subscribed to and the product they are accessing.

The role principals are just one type of principal that determines permissions for a user. Others may be determined at runtime based on the association between a user and an object or other internal logic of the object. A principal is an identifier assigned to a user actor, which may be a person, or external system, in the context of an authorization request.

Each user who belongs to a corporate account is assigned a unique identifier and at least one role principal before authorization can proceed. The user may also be associated with a resource that permits stronger or weaker permissions. This principal is defined as a user-type principal.

Principal types can include *role principals* (e.g., "manager"), *user principals* (e.g., "assigned-user"), *SSN principal* (e.g., "234-23-2332") or *access principal* (e.g., "wireless", "web"). The two principals currently supported by SSF are role and user, although it is easy to add more as needed.

The AuthorizationContext object must be created for each authorization request. Web based applications may create a subclass class that interacts with HttpRequest objects to obtain and cache user data in an encrypted cookie that is valid for the user's session.

com.hippikon.security.ProtectedResource

Any object within an application that needs protection by the authorization framework must extend from `com.hippikon.security.ProtectedResource`.

The ProtectedResource abstract class defines a single method:

```
public static String getResourceName() throws ProtectedResourceNamingException
```

Subclasses must override this method as the default implementation throws an automatic exception if invoked. The return value of this method should describe the class of ProtectedResource that can be used as a key in a PolicyStore implementation.

The return value is also used to bind Policy implementations to ProtectedResource classes at deployment. Because developers of future applications need to know the return value of this method explicitly so **this should be reflected in the javadoc documentation**.

Developers are advised not to return the classname using `this.getClass().getName()` because many objects may be developed that need the same protected. For instance, a business object named RFQ may provide objects such as SummaryRFQBean, DetailedRFQBean or RFQBean for efficiency in a J2EE application (see the Value Object design pattern). Each of these objects should be considered an RFQ and have the same permissions.

To avoid the need to create multiple or duplicate entries in a PolicyStore each business object should implement `getResourceName()` to return the same value or, preferably, extend from a class that itself extends ProtectedResource. Each class would then inherit the same return value from its superclass.

The permissions of a ProtectedResource may differ from each entry depending on the object hierachy of an application. For example, an RFQ may be defined as a parent object or a child object. In the latter case, the permissions may be affected by the parent.

com.hippikon.security.PermissionSet

The PermissionSet interface represents a set of actions a user is authorized to perform on a ProtectedResource. It is similar to the UNIX permission flags but defines five flags instead of three:

```
public boolean canCreate()
public boolean canRead()
public boolean canWrite()
public boolean canControl()
public boolean canDelete()
```

To prevent application logic from bypassing the authorization framework the PermissionSet interface provides no mutator methods of any kind, however the authorization framework implementation must be able to create new objects and manipulate the permissions as they are loaded from a PolicyStore.

To meet these requirements, the following classes are provided to Policy developers:

```
com.hippikon.security.DefaultPermissionSet
com.hippikon.security.MutablePermissionSet
```

Neither of these classes provides public constructors, thus preventing clients from creating PermissionSet instances and potentially altering the PermissionSet retrieved from a PermissionsFactory. The Policy abstract class provides some toolkit methods, which provide controlled hooks that allow developers to create specialized logic for a ProtectedResource object.

The PermissionSet interface should be used only for authorization logic and not business or application logic, although application logic may semantically match a PermissionSet returned from a call to the PermissionsFactory. Consider the case where an RFQ object has been versioned and a user views one of the versions. The application logic may enforce business rules that prevent version branching. Therefore, an RFQ version should never be edited.

The application should employ its own logic to ensure no edit UI widgets are presented as part of the view. The PermissionSet returned from the PermissionsFactory may also have the WRITE permission disabled through a specialized Policy subclass that is bound to the RFQ object through the resource.policies file.

In general, it is dangerous for an application developer to rely on the PermissionSet to determine a UI view, as there can be many reasons why a permission could be disabled.

com.hippikon.security.PermissionsFactory

The PermissionsFactory class provides five methods that clients use to obtain a PermissionSet for one or more ProtectedResource objects, Class or combination of both within an AuthorizationContext:

```
public static PermissionSet
getPermissions(List resources, AuthorizationContext ctx)
```

```
public static PermissionSet
getPermissions(List resources, Class c, AuthorizationContext ctx)
```

The PermissionFactory methods are coupled to PolicyStore entries for an application. Each application provides a top-level container or context for child ProtectedResource object types. The method arguments passed to the PermissionFactory must match the object hierachy defined within an application PolicyStore. It is the responsibility of the application developers to define the PolicyStore and ensure all other developers know the legal combinations of objects and classes.

When a client needs to obtain a PermissionSet for an object defined as a child of one or more parents, the methods that take a List of ProtectedResources should be used. When the client needs to check permissions for a class of ProtectedResource (to determine whether new instances may be created for example), the methods that take a Class as an argument should be used.

To provide a convenient API and remove the burden of creating List objects when there may only be one top-level ProtectedResource or Class, the PermissionsFactory provides the following utility methods:

```
public static PermissionSet
getPermissions(ProtectedResource res, AuthorizationContext ctx)
```

```
public static PermissionSet
getPermissions(ProtectedResource res, Class c, AuthorizationContext ctx)
```

```
public static PermissionSet
getPermissions(Class c, AuthorizationContext ctx)
```

All methods may throw either `com.hippikon.security.IllegalAuthorizationException` or `com.hippikon.security.ResourceNotFoundException`.

Although each method defined in the PermissionsFactory class is declared `static`, the methods have been design to be thread-safe with minimal synchronization in order to maximize performance and remove class-level locking. The class is, therefore, suitable for server-side applications.

com.hippikon.security.IllegalAuthorizationException

The IllegalAuthorizationException signals whether any access to a ProtectedResource is denied on the grounds that basic authorization data provided in the AuthorizationContext is either not present or invalid. Such a condition would include:

1. A user who has not been assigned at least one role
2. A user whose account is disabled
3. A user who belongs to an account with that has no access to a product (maybe disabled due to a missed payment)
4. A user belonging to an account that has a limit on the number of concurrent user sessions

The IllegalAuthorizationException should not be used for application logic but should generate a log entry or trigger a notification event. Any-time this exception is thrown a possible breach of security may be underway that should be investigated promptly.

Authorization Classes

com.hippikon.security.Policy

The Policy abstract class provides application developers with hooks into the white-box authorization framework and is the primary mechanism for framework reuse.

Policy subclasses should be created where user principals defined in an application PolicyStore must be determined at runtime for a ProtectedResource, and which may change depending on the application that hosts the object.

For example, an Document has a user principal defined in a publishing system as 'the-editor' with a PermissionSet of read, write and control. In another application this user principal may have no meaning whatsoever and 'the-creator' may take on a similar meaning.

The `com.hippikon.security.Policy` class is an implementation of the Strategy or Policy Design Pattern (Design Patterns: Elements of Reusable Object-Oriented Software - GoF), and it decouples authorization logic from the implementation of a business entity object.

The alternative to using a Policy Design Pattern is to embed this logic in the object itself. However, this alternative breaks the object-oriented paradigm because it breaks a cardinal rule of keeping related data and behaviour in once place. Not only would an object provide its own interface to accomplish one set of coheseive tasks, but authorization logic is now included within it, and authorization logic is littered across the application code. Another option would be to use Aspect Oriented Programming techniques, but this would add yet another layer of complexity. The only coupling we have with Hippikon is through the ProtectedResource class, which passes OO-type questions such as 'Is an BusinessDocument a ProtectedResource?' The callback methods allow user principals to be determined at runtime on a per instance or user basis. Further the callback methods allow specialized authorization logic that can alter the PermissionSet obtained from a PolicyStore to be modified in a controlled sandbox environment.

The class defines two callback methods and two toolkit methods previously mentioned in the description of the PermissionSet interface.

```
protected List determineUserPrincipals()
protected List doFinal(PermissionSet perms)
```

The first method will most likely examine the return value of the `getUserGUID()` defined in the `AuthorizationContext` and compare with some logic specific to the ProtectedResource type. The Policy for an RFQ may be implemented as:

```
protected List determineUserPrincipals() {
    List list = new ArrayList();
    if (ctx.getUserGUID() == document.getEditorUserGUID()) {
        list.add("the-editor");
    }
    return list;
}
```

When a List of ProtectedResource objects is passed to the PermissionsFactory, the framework determines which Policy subclass to instantiate by checking the resource.policies file for the product being accessed. It then invokes the `determineUserPrincipals()` method for each Policy in order to build up the complete list of principals for the AuthorizationContext; this includes the role principals and all user principals.

After all principals have been determined, each may be looked up in the PolicyStore created for the product order to obtain the complete set of permissions for a user.

Once the PermissionSet has been obtained for all principals, the final callback method `doFinal(PermissionSet perms)` is invoked in reverse order. This method allows Policy implementations to affect a change in the PermissionSet passed back to the client based on specialized logic. For example, the developer of the DocumentPolicy could turn off write, control and delete permissions if the Document object it protects was an immutable version.

The List of ProtectedResource names are made available to Policy subclasses by way of the `getResourcePathInContext()` method, which may be examined in the `doFinal(PermissionSet perms)` method should authorization logic needs to be implemented that depends on the object hierachy being accessed.

The `getResourcePathInContext()` method should be implemented by policies that are at the top of the object hierachy (i.e., the primary application container objects) rather than subordinate policies.

The `com.hippikon.security.DefaultObjectPolicy` class covers 70-80% of ProtectedResources for an application. This class handles all role principal lookup because it is available from the AuthorizationContext object passed in at runtime. The callback methods return an empty List of user principals and do not affect the PermissionSet passed into the `doFinal(PermissionSet perms)` method.

It is more likely that Policy subclasses will be created for top-level container objects for an application. These subclasses tend to be coarse-grained, key domain objects that may contain many different types of child objects.

com.hippikon.security.PolicyStore

The PolicyStore abstract class provides an abstraction of a set of entries mapping principal permissions to a ProtectedResource. It also defines the legal combinations or ProtectedResource object hierarchy for a application.

Each product must provide its own PolicyStore. The PolicyStore should be created by application developers who know about each ProtectedResource and the business rules of the application. The authorization framework currently supports an XML-based policy store definition. Each application must provide an XML file that contains permission rules named:

```
hippikon.product-id.[productID].policy-store.xml
```

where [productID] is the unique identifier assigned to each product by the developer. If a single application is being created, a productID of "1" will suffice. The XML policy file must be located on the system CLASSPATH for it to be loaded and parsed at runtime.

Each permission set is represented in XML by using UNIX-style descriptions. Letters or dashes can be combined for each principal entry. See the DefaultPermissionSet API for a complete list of available permission flags.

An entry for an Document with a user principal of *the-editor* granted read and write permissions would look like:

```
<protected-resource name="Document">
    <principal type="user" name="the-editor" acl="-rw--"/>
</protected-resource>
```

Permissions may be nested to any depth, and permissions inheritance avoids duplication and provide fine-grained permissions definitions. If a child object is added to the Document named *Note* and add a permission of read is added for users in a manager role, the XML entry looks like:

```
<protected-resource name="Document">
    <principal type="user" name="the-editor" acl="-rw---"/>
    <protected-resource name="Note">
        <principal type="role" name="manager" acl="-r---"/>
        <principal type="user" name="the-editor" acl="ir---"/>
    <protected-resource>
</protected-resource>
```

In this example, all users who are assigned a role of *manager* can read notes contained within all Document objects, but they can not read the Document itself. The user that is *the-editor* (when the Policy callback methods are invoked) is the only person who can create new instances of a Note attached to an Document.

Until a permission is explicitly defined in XML for a ProtectedResource, either an empty PermissionSet is returned or an IllegalAuthorizationExceptio is thrown, depending on the AuthorizationContext.

Permissions may also be overridden further down in an object hierarchy:

```
<protected-resource name="Document">
    <principal type="user" name="the-editor" acl="-rw---"/>
    <principal type="role" name="manager" acl="-r----"/>
    <protected-resource name="Note">
        <principal type="role" name="manager" acl="-----"/>
        <principal type="user" name="the-editor" acl="ir---"/>
    <protected-resource>
</protected-resource>
```

In the above example, the role of manager is granted read access to Document objects, but not all permissions are disabled for child notes.

Child objects may also inherit the permissions of their parents:

```
<protected-resource name="Document">
    <principal type="user" name="the-editor" acl="-rw---"/>
    <protected-resource name="Note">
    <protected-resource>
</protected-resource>
```

In this example, the note has the same permissions defined as the Document parent. Most of the work in defining a nested data structure and principal lookup is provided by a package visible class `com.hippikon.security.DefaultPolicyStore`. This class provides one load method that may be overridden by subclasses in order to populate the data structure from a persistant store.

The `com.hippikon.security.XMLPolicyStore` class is the default implementation for the current version, although it is completely hidden from clients. This allows the underlying store mechanism to be changed with no impact.

SDK Examples

Implementing specialized Policy classes

To plug Policy implementations into the framework, developers must follow these steps:

1. Extend ProtectedResource and implement `getResourceName()` to return a descriptive name for the resource. This name may be the classname of the business interface it implements. Document the return value in the javadoc comments
2. Map the return value of `getResourceName` to an entry in a PolicyStore
3. Define role and user principal permissions in the PolicyStore
4. Create a Policy subclass if required
5. Map the Policy subclass to the ProtectedResource in the resource.policies file
6. Provide a constructor that matches that of the Policy class. This constructor will narrow-cast the ProtectedResource object passed in by the framework to the specific type
7. Implement the callback methods as needed

Example

```
package com.hippikon.security.test;

import java.util.*;
import com.hippikon.security.*;

/**
 * A default implementation of an Policy for an Document
 */
public class DocumentPolicy extends DefaultObjectPolicy {

    private Document doc;

    /**
     * All Policy subclasses must provide a constructor with this signature.
```

```
      */
     public DocumentPolicy(ProtectedResource res, AuthorizationContext ctx)
     throws IllegalAuthorizationException {

         // must call the super constructor
         //
         super(res, ctx);

         // narrow-cast the ProtectedResource to the type we expect
         //
         this.doc = (Document)res;
     }

     /**
      * Determines if the user specified in the AuthorizationContext
      * is either 'the-editor' or 'the-author', 'the-reviewer',
      * or 'the-manager'
      */
     protected List determineUserPrincipals() {

         List list = new ArrayList();

         if (ctx.getUserGUID().equals(doc.getManagerGUID())) {
             list.add("the-manager");
         } else
         if (ctx.getUserGUID().equals(doc.getEditorUserGUID())) {
            list.add("the-editor");
         } else
         if (ctx.getUserGUID().equals(doc.getAuthorGUID())) {
            list.add("the-author");
         } else
         if (ctx.getUserGUID().equals(doc.getReviewerGUID()) {
            list.add("the-reviewer");
         }
         return list;
     }

     /**
      * Turns off write, control and delete if this isn't the latest version
      */
     protected PermissionSet doFinal(PermissionSet perms) {

         MutablePermissionSet mps = createMutablePermissionSet(perms);

         if (!doc.isLatestVersion()) {
             mps.setReadOnly();
         }

         return mps;
     }
 }
```

The resource.policies file

To invoke a Policy for a ProtectedResource, a binding must be made in the resource.policies file for each application. This file must follow the standard Java Properties file format and be located on the system CLASSPATH.

The file must be named similarly to the XML policy store file:

```
hippikon.product-id.[product-id].resource.policies
```

and contain entries that maps the return values of `getResourceName()` to Policy classnames. An example entry for a Document ProtectedResource that maps to `com.yourDomain.security.DocumentPolicy` looks like:

```
# map the DocumentPolicy to Document
#
Document.policy.classname = com.yourDomain.security.DocumentPolicy
```

If no binding is provided, the `com.hippikon.security.DefaultObjectPolicy` class is invoked. This class provides an out-of-the-box, working framework (Ref: The Selfish Class) that does not break an application if a developer forgets to include the resource file.

Policy Store Extensions Cache

To avoid the need to parse XML policy stores for each authorization request, the authorization framework implements a simple cache to store the policy store data structure in memory.

The default cache interval is 6 hours, although this may be altered by setting a system environment variable. The value must be defined in milliseconds:

```
policy-store-factory.cache-flush.interval=3000
```

The example above causes the cache to be flushed every 3 seconds. Care must be taken not to allow a setting as short of this in a production environment. The cache flush property can be set as an environment variable.

Permissions Navigator GUI

Creating complex authorization rules for an extensive object hierachy that contains many principals can be tedious for developers and difficult for test engineers and product managers to read. Understanding the authorization rules in any system can be difficult enough, but with a complex system combinatorics can explode.

To assit developers, testers and requirements specification writers Hippikon provides a user interface that provides a convenient display format for XML Policy Store files. With Hippikon 3.3 the interface provides XML Policy Store authoring functionality. Since the target audience for the Permissions Navigator is developers and testers, ANT is the platform dependent method to launch the GUI. Writing a wrapper script to launch GUI is trivial. The main() method is included in the file `com.hippikon.security.PermsNavigator`.

To launch the GUI, cd to the directory where you extracted the hippikon download and at a command prompt type:

```
cd [path to hippikon directory]
ant run-gui
```

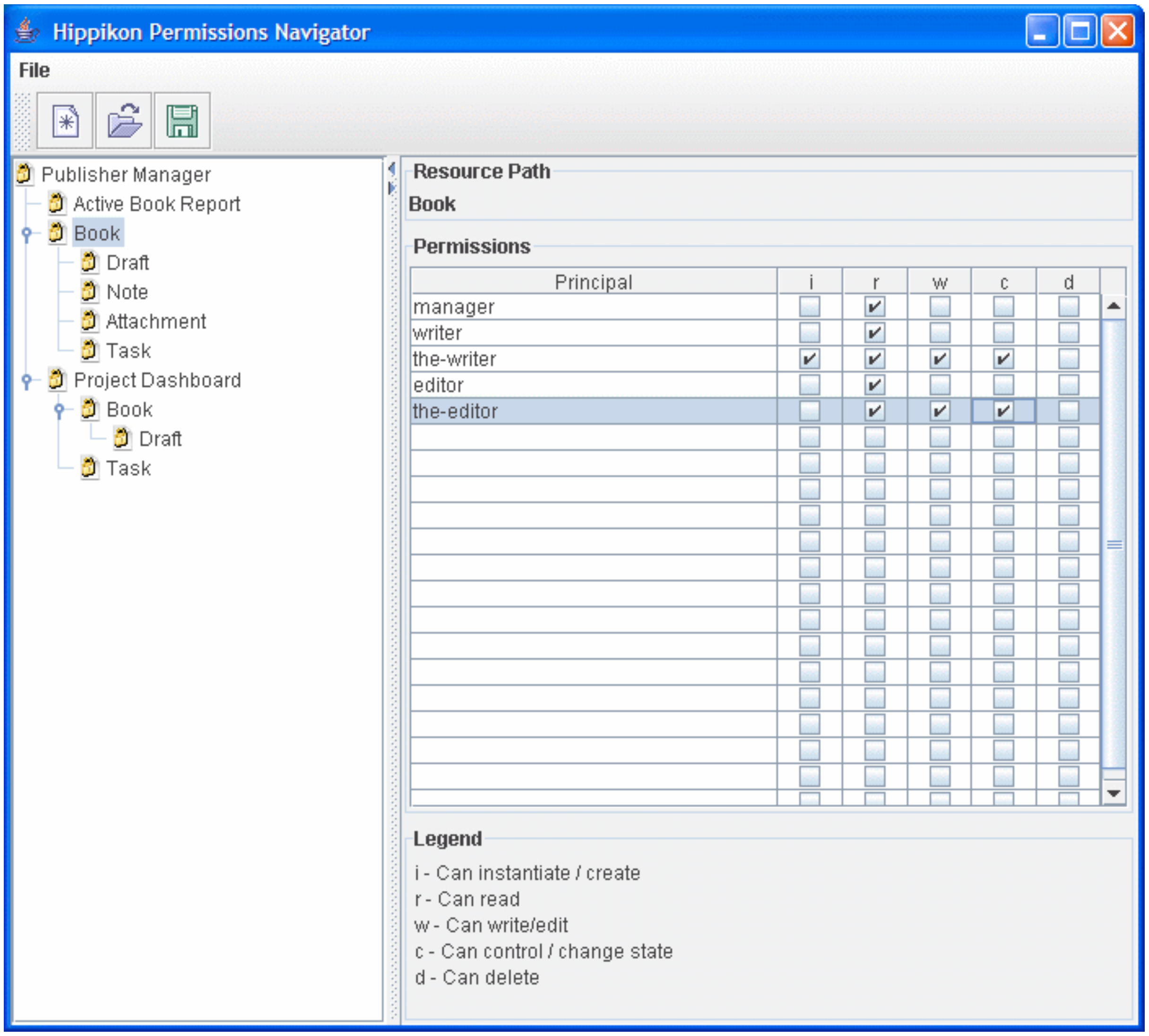
The following screen shots are of the Permissions Navigator running using the default Metal Look and Feel on a WinXP machine running JDK 1.5.

Viewing XML Policy Store Files

The Permissions Navigator provides a convenient view of an XML Policy Store file in the form of a tree and a custom spreadsheet. The tree on the left side of the user interface represnts the valid paths of protected resource within an application. The top-most node of the tree represents the host application itself.

The spreadsheet view on the left displays the principals and permissions defined for the selected protected resource node on the right. The view represents the exact same permissions that would be calculated by the Hippikon framework (although run-time Policy callback classes are not invoked).

The graphical view provides a convenient means for testers to find out what permissions are defined for a set of objects, and for developers to check their implementation against requirements. It may also allow Product Managers or Professional Services employees to 'configure' an application for a customer without needing development input. The possibilities are widespread, especially for enterprise software where highly customized software is the accepted norm.



Resource Path

Book

Permissions

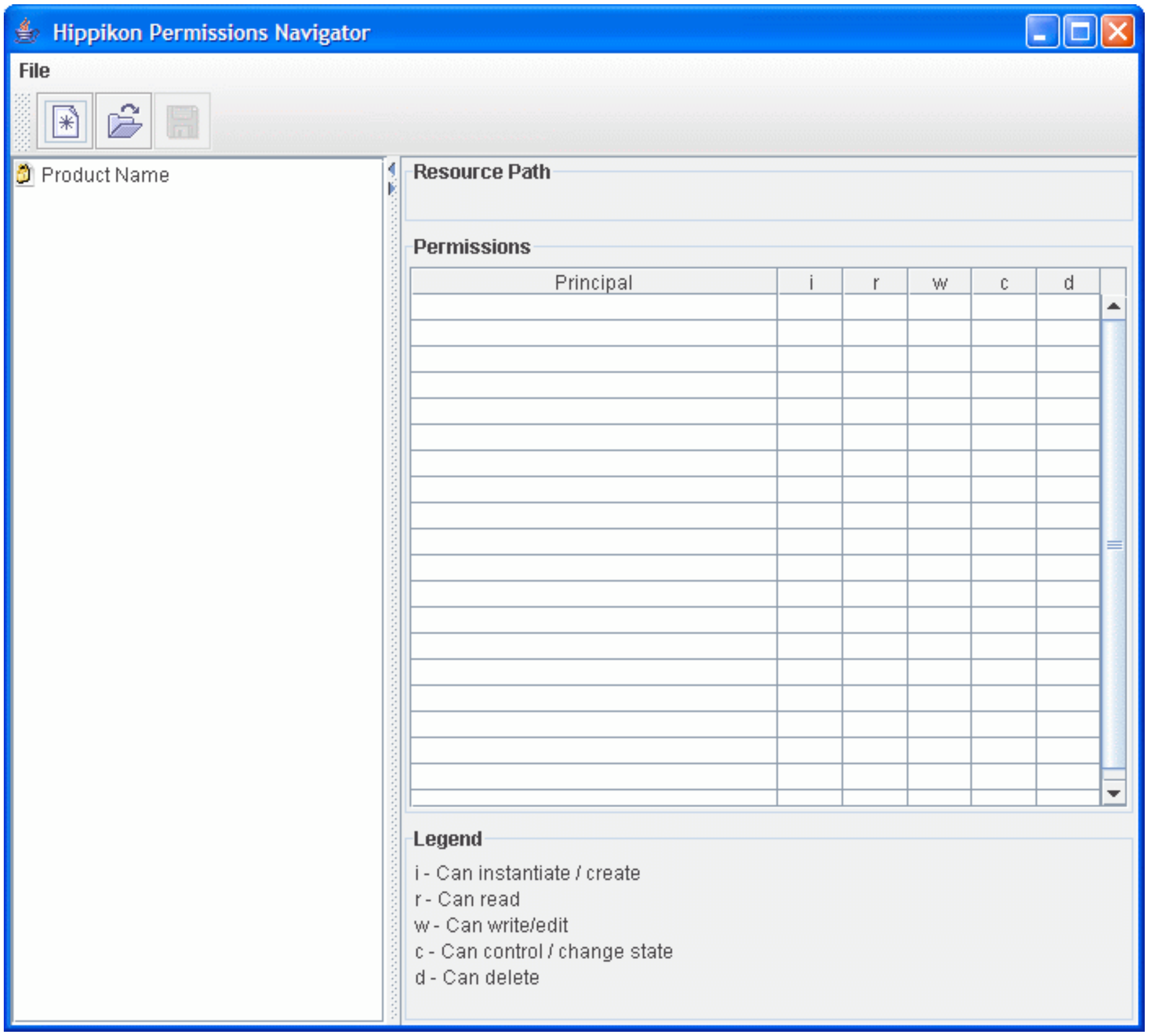
Principal	i	r	w	c	d
manager	☐	✓	☐	☐	☐
writer	☐	✓	☐	☐	☐
the-writer	✓	✓	✓	✓	☐
editor	☐	✓	☐	☐	☐
the-editor	☐	✓	✓	✓	☐

Legend

i - Can instantiate / create
r - Can read
w - Can write/edit
c - Can control / change state
d - Can delete

Creating XML Policy Store Files

The first step in creating a new XML Policy Store file is to create a new document by choosing File->New Policy Store or clicking on the Blank Document icon on the toolbar.

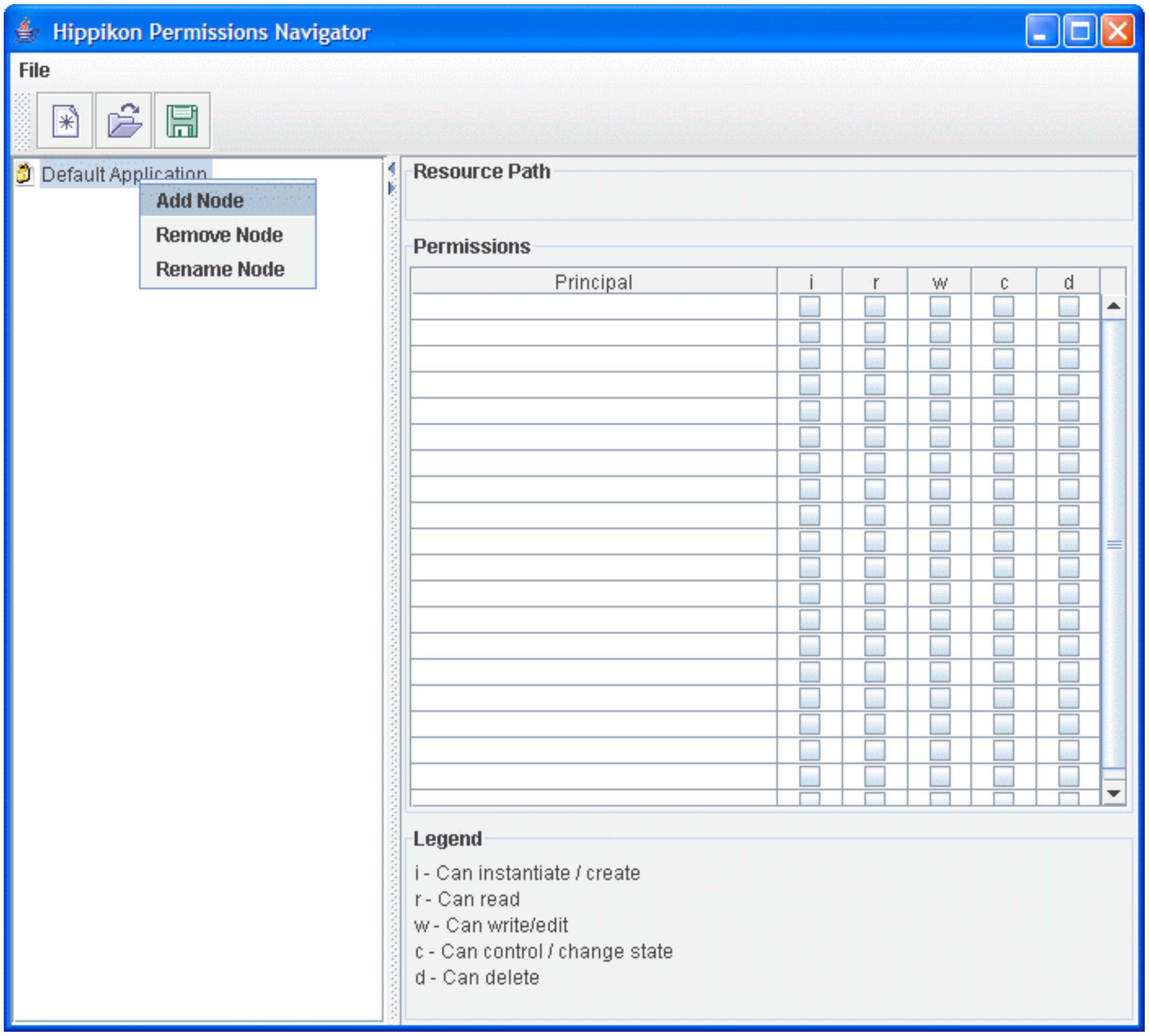


Principal	i	r	w	c	d

Legend

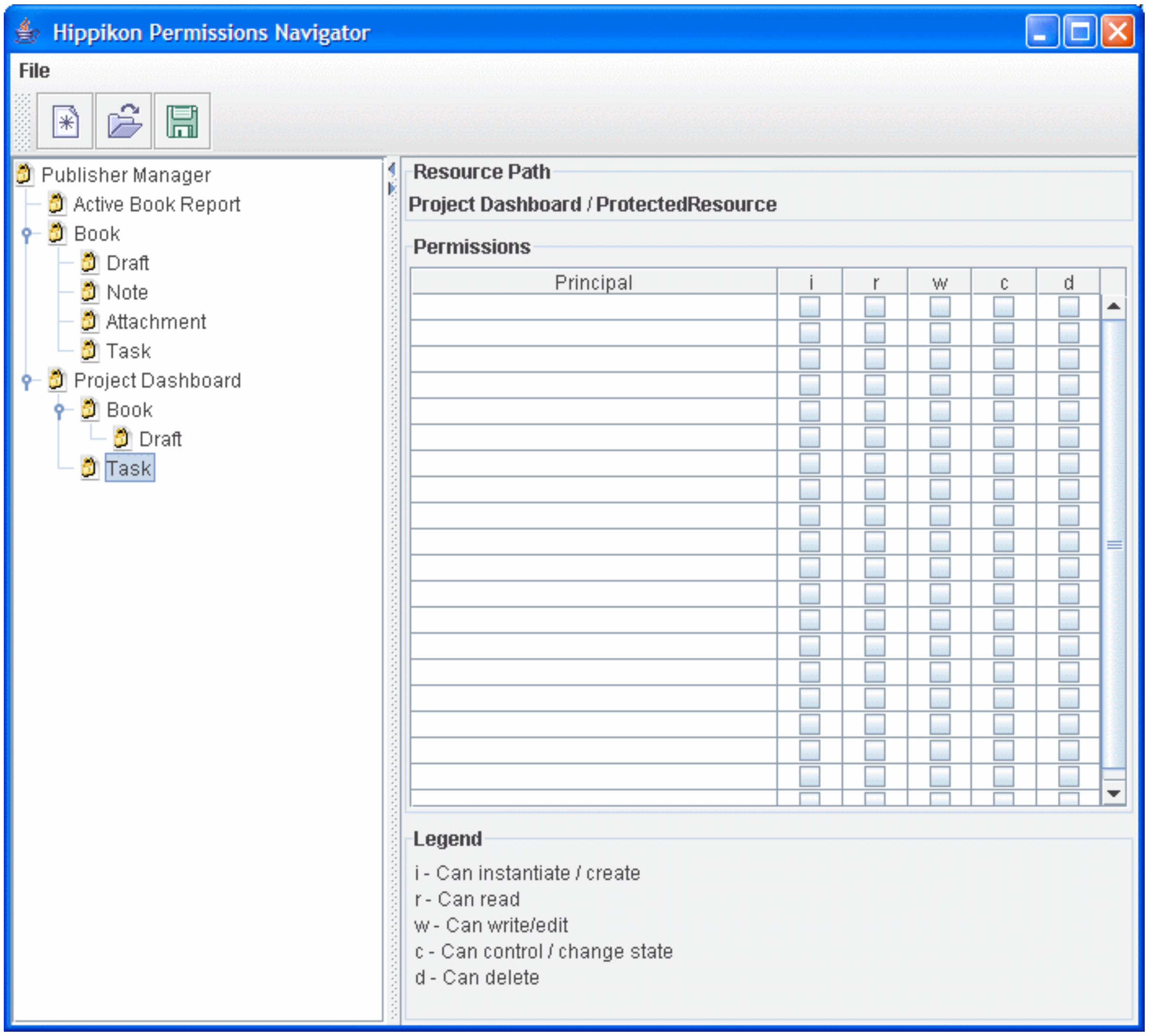
i - Can instantiate / create
r - Can read
w - Can write/edit
c - Can control / change state
d - Can delete

You can then start by adding nodes to build up the object hierachy. Each node in the tree displayed on the left hand side represents a valid resource path in the target application.



i - Can instantiate / create
r - Can read
w - Can write/edit
c - Can control / change state
d - Can delete

Once nodes have been added they can be renamed by right-clicking on the node and choosing 'Rename Node' or by simply clicking once on the node name.



Resource Path

Project Dashboard / ProtectedResource

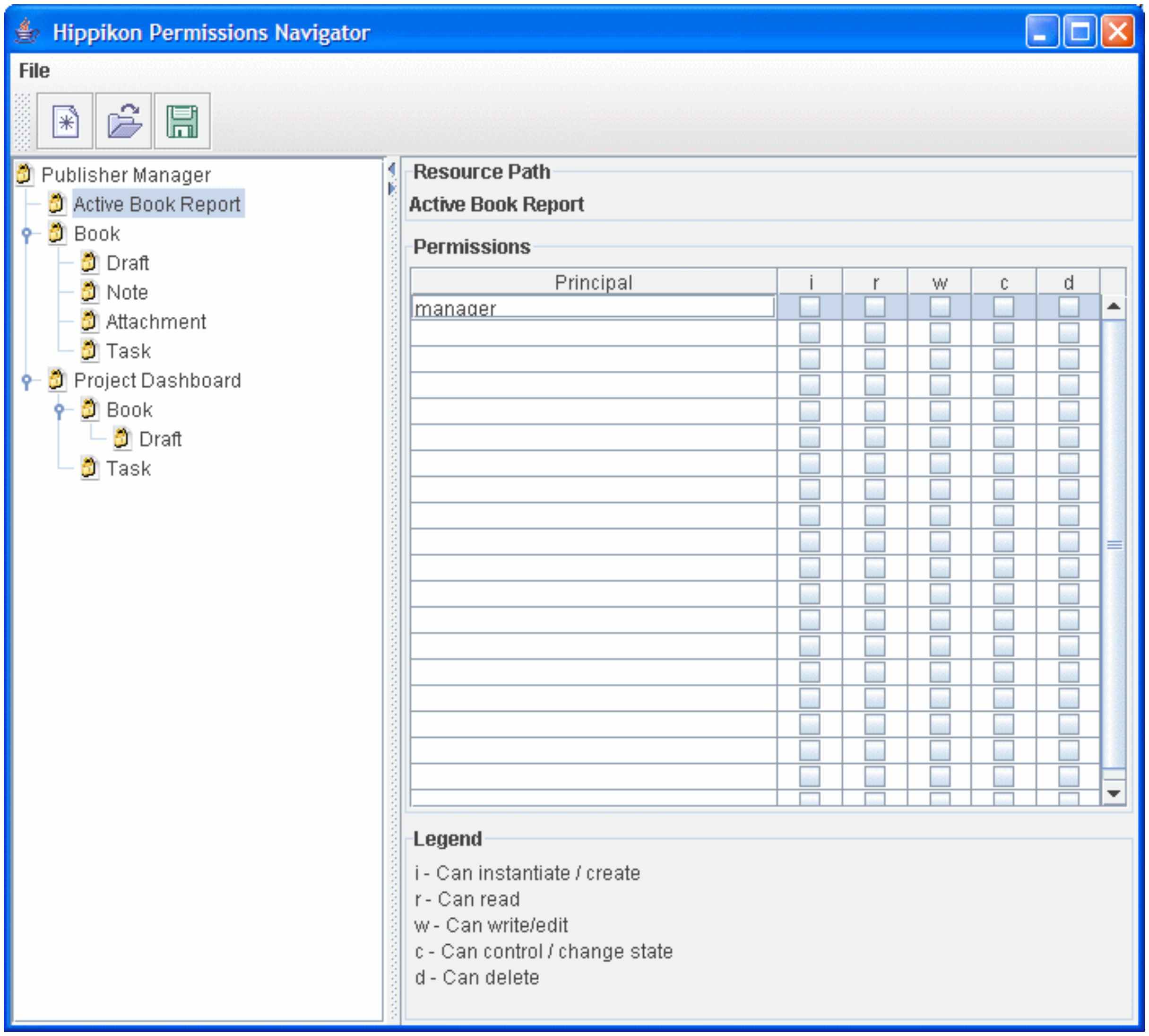
Permissions

Principal	i	r	w	c	d

Legend

i - Can instantiate / create
r - Can read
w - Can write/edit
c - Can control / change state
d - Can delete

Once the nodes have been added the principals can be defined. Note that principals may not be added to the top most node, which represents the host application container for all proteced resource objects. The GUI will not allow principals to be added to the top node to avoid confusion. To add a principal just double-click in the Permissions table display on the right and type the principal name. Duplicate names are not allowed on the same node.



Resource Path

Active Book Report

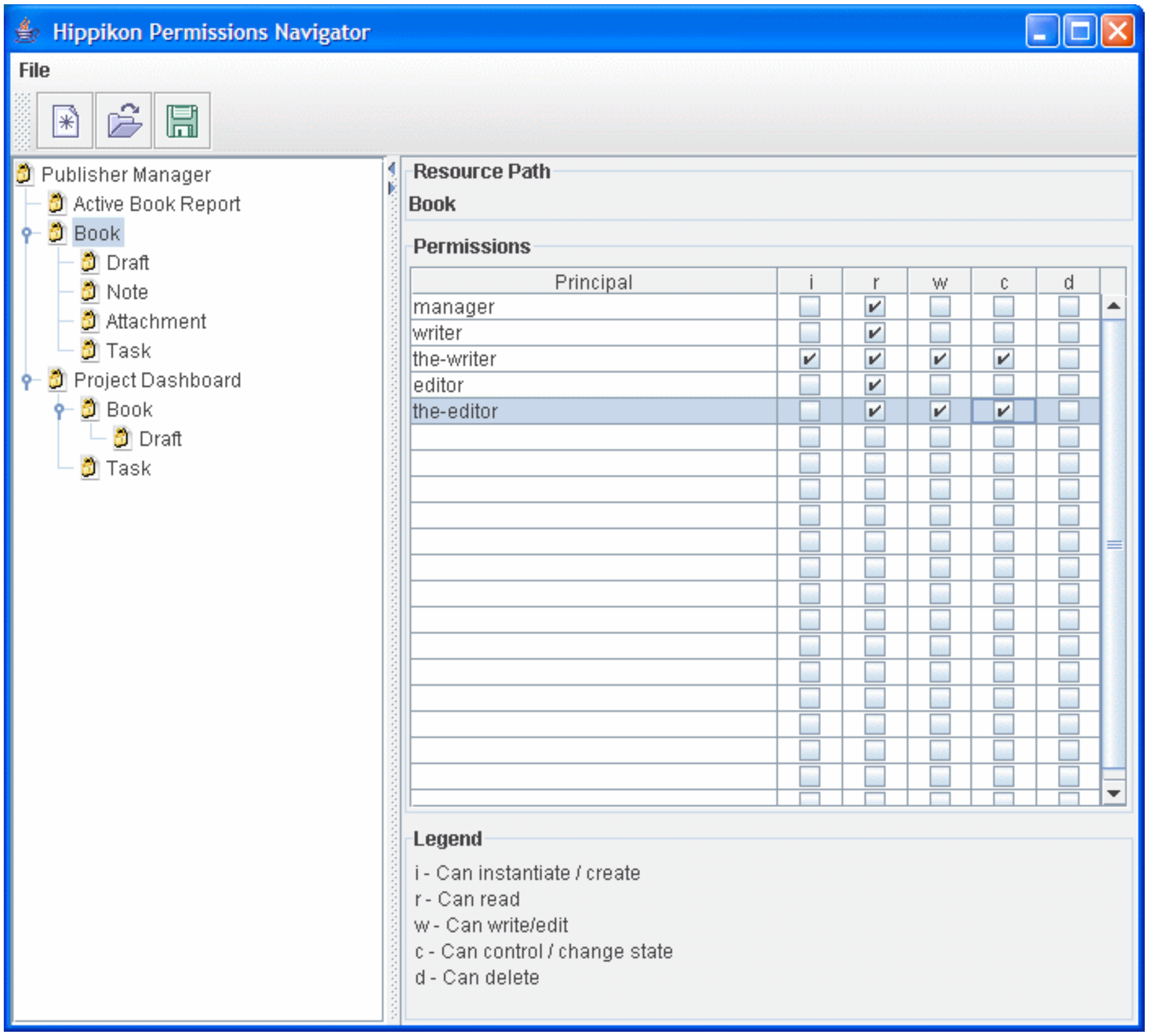
Permissions

Principal	i	r	w	c	d
manager	☐	☐	☐	☐	☐

Legend

i - Can instantiate / create
r - Can read
w - Can write/edit
c - Can control / change state
d - Can delete

Once principals have been added permissions can be assigned by checking the appropriate checkbox in the I, R, W, C or D columns in the right hand table. It should be noted that if a principal has a checkbox left unchecked that permission is set to '-' meaning the permission is denied rather than undefined.



Resource Path

Book

Permissions

Principal	i	r	w	c	d
manager		✓			
writer		✓			
the-writer	✓	✓	✓	✓	
editor		✓			
the-editor		✓	✓	✓	

Legend

i - Can instantiate / create
r - Can read
w - Can write/edit
c - Can control / change state
d - Can delete

If you define a set of principals and permissions for a node that has child nodes, those defined principals and permissions will be inherited by the child nodes further down the tree. In this way you can define principals easily, but also override the rules further down the tree. For instance, if the principal 'manager' was given create, read, write, control and delete permissions to the node 'Book', the nodes 'Draft', 'Note', 'Attachment' and 'Task' would all inhert the same set of permissions. However, if you needed to set the permissions for the 'Note' child node for the manager to read-only you could do so by simply clicking on the 'Note' child node of 'Book' and uncheck create, write, control and delete checkboxes for the manager principal. In this way very complex rules can quickly be created for an application.

Once the rules have been defined choose File->Save As or click the floppy disk icon on the toolbar to save your file to disk. The GUI will convert the tree and table structure to a valid XML document that can be used within the hippikon framework. To verify your file you can load it back into the GUI by choosing File->Load Policy Store or clicking the folder icon on the toolbar.