# Table of Contents

## Part I Introduction

1. Welcome to the Ignition SDK .......................................................... 6  
2. What can you do? ................................................................. 6  
3. Prerequisite Knowledge ......................................................... 7  
4. Technical Documentation ....................................................... 7  
5. Getting Help ................................................................. 7  

## Part II Getting Started

1. The Ignition Platform ......................................................... 10  
2. SDK Structure ................................................................. 11  
3. Development Cycle ......................................................... 12  
4. Environment Setup ......................................................... 12  
5. Anatomy of a Module ....................................................... 13  
6. Scopes, Hooks, and Contexts ............................................ 15  
7. Useful Utility Classes ....................................................... 15  
8. Key Design Concepts ....................................................... 17  
    - Overview ................................................................. 17  
    - Logging ................................................................. 17  
    - Storing Configuration Data ........................................ 18  
    - Extension Points ..................................................... 19  
    - Programming for Redundancy ..................................... 19  
    - Localization ............................................................ 20  
    - Style and Naming Conventions .................................. 21  

## Part III Programming for the Gateway

1. Getting Started - The GatewayModuleHook ................................ 24  
2. Providing Status Information ........................................... 24  
3. Protecting Modules with the Licensing System ....................... 25  
4. Storing data with PersistentRecords ................................... 26  
5. Projects and Project Resources ......................................... 29  
6. Extending Ignition with Extension Points ............................ 31  
7. Storing History using Store and Forward ............................ 32  
8. Working with Platform Services ....................................... 33  
    - Databases ............................................................... 33  
    - Execution Scheduling ............................................ 34  
    - Auditing ................................................................. 35  
    - OPC ................................................................. 36  
    - SQLTags .............................................................. 37  
    - Alerting ............................................................... 38
## Part IV  Programming for the Designer  

1. Working with Project Resources ................................................................. 42
2. Building a Workspace .................................................................................. 44
3. Creating Menus and Toolbars .................................................................... 45
4. Supporting Search and Replace ................................................................. 46
5. Supporting Undo and Redo ....................................................................... 47
6. Designer to Gateway Communication (RPC) ........................................... 48
7. Gateway to Client Communication (Push Notification) ............................. 49
8. Providing Progress Information ................................................................ 51

## Part V  Vision Component Development  

1. Module Structure ....................................................................................... 54
2. Component Overview ................................................................................ 54
3. BeanInfo Classes .................................................................................... 55
4. Designer Hook ......................................................................................... 56

## Part VI  OPC-UA Driver Development  

1. Three Steps to a UA Driver ........................................................................ 60
   - Implement the Driver Interface ............................................................... 60
   - Implement the DriverMeta Interface ...................................................... 60
   - Register Your Driver ........................................................................... 61

## Part VII  Examples  

1. Component Example ............................................................................... 64
2. SQLTag Provider Example ....................................................................... 64

Index  

© 2011 Inductive Automation
Introduction

Part I
1 Introduction

1.1 Welcome to the Ignition SDK

Ignition was designed to be a powerful industrial application platform, built from the ground up to support extensibility through a modular architecture. In fact, nearly all of the commercial pieces associated with Ignition, such as SQLBridge, Vision, and OPC-UA, are modules built with the same SDK that you have in your hands.

As a platform, Ignition provides a variety of services that dramatically reduce the time and effort required to develop top-quality industrial applications. Instead of worrying about the OPC specification, or how to efficiently handle database connection pooling, you can focus on achieving your main goals. Using the SDK, you'll be developing modules that are loaded into the platform and given access to everything it can do. In addition to using these services, in many cases modules can extend them to provide additional implementations, such as adding a new way to authenticate users. Modules can provide new components for Vision screens, create new workspaces in the designer, and add drivers that expose data through OPC-UA. In short, if you can dream it, it's likely the Ignition platform can help you do it.

This document is written to be a friendly reference guide to the platform, and to provide all of the information you need to get started with the SDK. Certainly some parts of the SDK might not be covered as well as they could be, and we always encourage feedback on this point, though when coupled with the technical documentation (the JavaDocs) everything should at least be represented. As questions come up, or you come up with ideas for way the platform could be improved, we encourage you to contact us, preferably through the Module SDK Forum on our website.

We hope that you will find programming for Ignition as exciting and fun as we've found the process of building it, and look forward to seeing the great things you create with the SDK!

Happy programming,

The Inductive Automation Development Team

1.2 What can you do?

An Ignition module can do anything you might imagine. The scope of an Ignition module is extremely broad - from adding something minor to complement the framework, or leverage the framework to create an entirely new product. Modules can even add new functionality to other modules!

Here's some ideas, in no particular order, to help spur creativity:

- Write an ethernet driver for a PLC or other device.
- Add a visual component to the Vision module.
- Incorporate an existing 3rd party Java library to include some new technology
- Insert new useful functions into the scripting system
- Create a new authentication profile type
- Add some sort of industry-specific suite of functionality
- Implement a numerical analysis that Ignition doesn't yet include
1.3 Prerequisite Knowledge

To be successful with the Ignition Module SDK, you'll need to be acquainted with a few core programming technologies and concepts.

Java SE
Java is the language that Ignition is written in, and the language that modules must also be written in. Therefore, to even get started with the SDK, you'll need to have a decent foundation in the Java language and framework. Concepts such as package management, classpaths, and JAR files are frequently used in the development of Ignition modules.

See also:

Ant
Ant is build file system used by Ignition. While not strictly necessary for module development, understanding and using Ant will make things much easier. Each example project includes an Ant build file which can be modified and adapted as necessary.

See also:
Apache Ant homepage: http://ant.apache.org/

Ignition
Of course, to build an Ignition module, it's helpful to have good background in working with Ignition itself. You should have a good understanding of how many of the platform services work and are configured through the gateway, such as database connections, authentication profiles, SQLTags providers, OPC connections, and projects. For some targeted project types, such as a device driver for OPC-UA, a wide knowledge of the platform isn't necessary, but you should still be acquainted with how the existing drivers are configured and used.

1.4 Technical Documentation

In addition to this document, another crucial piece of information for developers working with the SDK is the JavaDocs for the API. That library, compiled off of the source code, provides information about all of the classes and interfaces in the API. The information in this document will often refer to code objects that will be further documented in that system.

Additionally, it's useful to consult the JDK documentation provided on Oracle's Java portal. When doing this, however, remember that the gateway is compiled against Java 6, but designer/client code is compiled against Java 5, so take care to only use the facilities appropriate to the scope of module you're developing.

1.5 Getting Help

If you run into a problem with the SDK, or aren't sure how to accomplish something, the module SDK forum on the Inductive Automation website is the place to go! There are a wide range of users who have experience building modules, and it is frequently checked by the IA development team.

For more general questions with Java, Ant, etc., the internet has a wealth of resources. StackOverflow.com is a particularly good place to ask questions, though you generally can't go wrong typing nearly anything into Google.
For developers who would like to jump-start their dev process, Inductive Automation hosts periodic Module Development Bootcamps, and can also host private training events. More information about these events can be obtained by calling the sales department at 800-266-7798.

Resources
Inductive Automation Forum: http://www.inductiveautomation.com/forum
Email: developers@inductiveautomation.com
2 Getting Started

2.1 The Ignition Platform

The "Ignition Platform", as we refer to it frequently throughout this document, is the base of the Ignition product and provides the groundwork upon which modules, including your own, can be built. When we refer to the platform, we are referring primarily to the publicly accessible functions that can be used by modules. Occasionally, though, we simply use it as a convenient way to describe Ignition independently from any particular modules.

The various sections and chapters of this guide will outline the functionality provided by the platform, first at a high level, and then further in depth. Despite the fact that we refer to the Ignition Platform as a single unit, there are really two distinct components to it: the gateway portion, and the designer portion. There is a third (as we refer to them later, “scope”), the client, but compared to the first two, its role is minimal. The structure of this document clearly reflects this main division, with separate chapters for programming for the gateway and for the designer. Understanding this structure will help you to better understand the layout of the SDK, with libraries generally split between Gateway, Designer, and Common.

The role of the SDK is to provide access to the platform through an API (application programming interface). In simple terms, the API consists of code that is both understood by Ignition, and public for use in your code. It is the "common layer" between what you write, and the private inner workings of the Ignition system. Through the course of this document, you'll learn about various interfaces and classes available in the API, and how you can use them to build modules.

The following diagram gives a visual overview to the services and managers available in the gateway.
2.2 SDK Structure

The SDK has the following folder structure:

/api
The JAR files that define the actual API. Outlined below.

/examples
Example projects that illustrate different types of modules.

/javadocs
The JavaDocs technical documentation compiled off of the API.

API JARs
The API consists of the following JAR files that will need to be referenced by your projects:

Core
client-api.jar
Needed for "client" scope
designer-api.jar
Needed for "designer" scope
gateway-api.jar
Needed for "gateway" scope

Vision
vis-client-api.jar
APIs exposed by the Vision Module
vis-designer-api.jar
vis-common.jar
Drivers
driver-api.jar

Common files for both the client and designer
APIs exposed by the OPC-UA module
Used to create new drivers for the OPC-UA module.

2.3 Development Cycle

The process of creating an Ignition module generally follows the following path:

1) Register for a developer key
   Normally, Ignition will only load modules that have been cryptographically signed by Inductive Automation. This is to prevent unauthorized code access to the platform, and prevents malicious code from masquerading as a legitimate module. Since this would make development of modules very tedious, as each build would need to be sent to IA for signing, the system can be activated in "developer mode" to allow the loading of unsigned modules. Developer keys can be obtained easily from the Inductive Automation website.

2) Download the SDK
   Since you're reading this, you've probably already done that.

3) Set up your development environment
   Any Java development environment can be used, but a setup based on Eclipse is outlined in Environment Setup.

4) Develop and Test
   As you write your module, you'll be able to compile it and hot-deploy right from Eclipse to your local Ignition installation that is in "Developer Mode". This allows you to test your module as you write it before sending it to us.

5) Send to Inductive Automation for signing
   Once you're ready to release your module, it must be signed by Inductive Automation. This can be done automatically through the developer portal on the website.

2.4 Environment Setup

Download the Java SDK

Download Eclipse

Set up a workspace, import the skeleton project

Install the Java JDK
   Download and install the latest Java 6 SE JDK (Java Development Kit). You can download this from Oracle's website. You'll need the JDK because it includes the java compiler, javac.

Download Eclipse
   Download Eclipse from http://www.eclipse.org. You want the download: "Eclipse IDE for Java Developers". This is a zip file - there is no installation. Unzip it somewhere meaningful (maybe C:\Program Files\Eclipse). To run Eclipse, you simply double-click on eclipse.exe, but wait!
First you need to set up your workspace.

**Set-Up Your Workspace**

When you run start up Eclipse, it asks you what workspace you want to work in. A workspace is a folder on your hard drive that holds a collection of projects. You can have multiple workspaces. If you're already an eclipse user, you'll want to create a new workspace.

To make a workspace, simple create a folder somewhere to be your workspace. For example: \development\IgnitionSDK_Workspace. Now unzip the SDK's examples into this folder. The examples in the SDK are set up to be ready-made Eclipse projects.

**Start Eclipse and Import Projects**

Now start eclipse by double-clicking on eclipse.exe. Point it to the folder you made as your workspace. If it starts with its friendly "intro screen" click on the "Go to the Workbench" button.

You'll notice that the Project Explorer is empty. Even though you unzipped the example projects into your workspace folder, Eclipse still wants you to "import" them. Click on File > Import... then choose "Existing Projects Into Workspace" and click "Next >". Browse for your workspace directory to fill in the "root directory" field. All of the example projects should appear. Make sure they're all selected and that "Copy projects into workspace" is not selected. Then click "Finish".

**Compile and Deploy**

Now all of your projects should be in Eclipse and it will compile (or "build") all of them. You'll notice that one of the projects is called "Build". This project has all of the Ant scripts that are used to compile and assemble the Java code into an Ignition *.modl file. Each project has its own build file. For example, the component example projects have a build file called build-component-example.xml. Double click on this Ant file to open it in Eclipse's Ant editor. (If it opens in a raw XML editor, close that, and then right click on it and choose Open With > Ant Editor). You'll see in Eclipse's Outline view each of the Ant script's Targets. These are what you'll use to compile your module. You can right-click on the "build" target and choose Run As > Ant Build to build that module file. If you have a Developer Mode Gateway running locally, you can run the "deploy" target to push your newly built module to your Gateway.

That's it! You're up and running. You've compiled and deployed a module using the SDK. Now you can write your own module or adapt one of the examples to create new functionality for Ignition.

### 2.5 Anatomy of a Module

In terms of the end product, a module is a *.modl file that is loaded into the gateway. That file, in reality, is simply a zip file containing all of the module's code (*.jar files) and a required manifest file that describes it. Each module is cryptographically signed in order to prevent unauthorized tampering, and Ignition will only load valid signed modules into the platform (except when in development mode).

In regards to code, a module can be made up of any number of projects that are compile into one or more JAR files. Somewhere in the code there will be Hook classes defined that allow Ignition access to the module. In addition to the module code, the JAR files might also contain resources such as images, and resource bundle files for localization.
The Module File

The module descriptor file

The module descriptor file is an XML file included in the root of the module package that defines a variety of properties about the modules. This file must always be named module.xml. A single descriptor can define multiple modules. For each module, the descriptor lists the JARs to be used, and their scopes. It also defines the minimum required platform version required, and the full path of the hook classes for each scope.

The module descriptor file can also define dependencies on other modules. For example, a module that adds components to Vision will require that module before it can be loaded.

Each of the examples includes a sample descriptor file.

File contents

The module package file should only contain the compiled JARs for the module and the descriptor file. The Ignition platform will load the module, locate the hooks, and start them up. Additional resources, such as images, should be compiled into the JAR files, and not included directly in the module package.

The Module Code

Project Structure

Ultimately, the structure of your module project (or projects, as most modules will consist of multiple sub-projects) is up to you. However, there are a few common structures that are used in most module scenarios. Primarily, breaking the projects up by scope, with another "common" project, is useful. Each project usually represents a JAR in the build process, and the common jar can be marked as applying to all scopes. Of course, this can also be accomplished through package structure and handled in the build process. The important point is that the resulting JAR files should be carefully tailored to their scopes, to reduce the amount of code unnecessarily loaded into scopes which don't use it.

Localization Support

The chapter on localization will discuss this topic more in depth, but in short, all string data that might be presented to the user should be held in external "resource bundle" files. There are several schemes for managing these files, but the easiest is simply to have a single file for each scope in the root package of your module and to load it into the BundleUtil in the hook. The section on Localization has more information about how this system works.

Design Best Practice

Strings that represent identifiers, names, etc., such as the "module id", should be defined in a single place in the common project, instead of repeatedly in code. For example:

Common Module Code

```java
public class ModuleMeta {
    public final static String MODULE_ID = "mymodule";
}
```

Now, all parts of your module can access the static values, and you don't have to worry about mis-typing an identifier.
2.6 Scopes, Hooks, and Contexts

Ignition defines three different scopes: the gateway, the designer, and clients. The scopes are used in various ways, such as defining which JARs should be loaded by each piece, and the addressing of push messages. A module may include pieces for multiple scopes, and resources in a module can be assigned to multiple scopes at once.

For each scope that the module wishes to run in, it will define a module hook. This is the entry point for the module in that scope, and provides several lifecycle functions. Each scope provides a context to the hook, which can be used to access all of the platform services of that scope.

Module Hooks

There are three interfaces that define the hooks, that correspond to the scopes: GatewayModuleHook, DesignerModuleHook, and ClientModuleHook.

In practice, it is usually best to extend from the Abstract implementations of these hooks (for example, AbstractGatewayModuleHook) instead of implementing the interfaces yourself.

Each module hook must be defined in the module manifest file.

Contexts

The context for the given scope will be passed to the setup or startup method of the associated hook. The modules should hold on to these contexts and pass them to subsystems that they define, as it is the primary way to access the services provided by the Ignition platform.

2.7 Useful Utility Classes

In the process of writing a module, you will likely encounter certain situations multiple times, including situations that have been faced many times by the developers on Ignition as well. For these very common cases, it’s useful to be aware of various helpful “utility” classes that exist in the platform that can help.

Gateway

TypeUtilities

The necessity of coercing values to a certain type is extremely common. The TypeUtilities class is extremely useful, and supports a large number of coercion scenarios. It goes far beyond basic casting, and can translate between many different types intelligently. For example, it can convert an enum to integer, first looking for a "getIntValue" function on the enum, and then using the ordinal position, and vice-versa.

DBUtilities

Provides several methods for easily closing connections, statements, result sets, etc. without having to perform all of the normal fault checking. For example, this common code:

```java
try{
    con = openConnection();
    ...database stuff and error handling...
}finally{
    if(con!=null){
```
try{
    con.close();
}catch(Exception ignore){
    //Not used;
}
}
}

becomes simply:
try{
    con = openConnection();
    //database stuff and error handling...
}finally{
    DBUtilities.close(con);
}

CloneUtil
Provides several different ways to clone objects, and a simple method that will detect which to use.

CRCUtil
Makes it easy to calculate the CRC of an arbitrary collection of objects, including a byte array.

FormatUtil
Provides function for formatting values into strings. In particular, has a number of functions for displaying durations and timing based messages.

GatewayUtils
Most of the functions in this class are only used by the Ignition platform, but there are some useful general-purpose functions for working with directories and manipulating files.

HexUtils
Tools for converting to and from hex strings into bytes.

SecurityUtils
Provides capabilities for encryption, decryption, and hashing.

Driver API
These byte utility classes are provided by the driver API and are very useful when writing protocol based drivers.

ByteUtilities
BCDByteUtilities

Designer

EDTUtil
Provides convenience functions for working with the EDT, and a useful coalescingInvokeLater() function that can be used to group multiple calls into a single event.

ErrorUtil
Many functions for displaying errors, but also other dialogs such as info and warning messages, and prompts.

IconUtil
Used for retrieving icons provided with Ignition.
2.8 Key Design Concepts

2.8.1 Overview

Before jumping into the technical aspects of programming for the gateway, client, and designer, there are a few core concepts that should be addressed at a high level. This chapter will introduce certain design paradigms that are common across all of Ignition, and may be different than other systems that you've programmed for in the past.

Most of the topics covered in this chapter will be discussed more completely later, but we felt it was important to provide a conceptual overview as soon as possible.

2.8.2 Logging

Ignition uses the Apache Log4J framework to record events, errors and debug messages. Log4J is a simple framework that makes it easy to log information, and store those logs in different formats.

Quick Introduction to Log4J

Log4J is a logging system, meaning that information is stored to it, recording what happened in the system. The information is stored to a particular Logger, which has a name, with a certain Severity rating. Appenders in the system receive the messages, and do something with them—such as writing them to a text file. Appenders can be configured to only log messages of a certain severity, and at any time we can set the level of severity that a given logger will accept by using that logger's name. For example, trace severity, which is the lowest, is usually very verbose, and not logged by default. If we are trying to troubleshoot a particular part of the system, and know which logger it uses, we can go and turn on trace logging for just that one, in order to see all of its information.

Getting a Logger

You can obtain a "Logger", which is a named entity used to log messages, by calling the static LogManager.getLogger functions. There are two primary methods used to get loggers—by class, and by name. The class method simple uses the full path of the class, whereas the name method allows you to use any name you'd like. For example:

```java
Logger log = LogManager.getLogger(getClass());
log.info("System started.");
```

or

```java
Logger log = LogManager.getLogger("MyModule.CoreSystem.Status")
```

Note that in Log4J, names are separated by ".". With the second example above, for instance, if we had other loggers with similar names under "CoreSystem", we could set them all to log debug messages as well by setting that level directly on "MyModule.CoreSystem".

Naming Loggers

While it is easy to get loggers based on class name, it is important to remember that these names will have little meaning to customers and integrators trying to use your module. On the other hand, using completely custom names sometimes makes it difficult to later track down where things are occurring in code. Still, it is usually advisable to use custom names that have meaning to both you and the end user, in order that someone who is trying to troubleshoot a problem can find the logger without help.

Logging Severities and Messages
Logging information is an art, and one that is difficult to perfect. There are a few elements that play into the concept of what "good logging" is:

- Log messages should contain helpful and identifiable data. For example, if you have a logger in a driver and write "Device connected", it will be of little use- multiple instances of a driver might be running at once, and the message does not indicate which instance it is referring to.
- Log messages set to Info and above should not "flood the log"- that is, report very frequently, making it impossible to see other logs, or filling up the allocated buffer quickly. There are several strategies for avoiding this, such as logging once, and then logging subsequent messages on Debug for a period of time, or only logging the first event, and then making the status easily visible in the gateway or designer.
- Log messages involving an exception should always include a custom message, in addition to the exception object. If only the exception object is provided, some appenders will not store the stack trace, making troubleshooting very difficult.

**Severities**

The use of severities is ultimately up to you, but here is a general guideline:

- **Error** - An error in the system, usually based on some sort of exception, that is unusual, and usually should be reviewed by somebody.
- **Warning** - Like error, an unusual event that should likely be reviewed. May indicate that something is not quite correct, but is not necessarily preventing correct operation.
- **Info** - Standard messages indicating a change of state, or anything else that might be beneficial for the user to know.
- **Debug** - Information used for troubleshooting, perhaps logged repeatedly or more frequently, provides technical information that might only make sense to a trained user or you, the developer.
- **Trace** - Very fine grained information that might be very verbose. Generally only logged for a short period of time in order to gather information for troubleshooting, this level is usually very technical, such as packet contents and result codes.

### 2.8.3 Storing Configuration Data

There are several ways to store configuration data (as opposed to process data) in Ignition, depending on what the data represents, and how it is used. The majority of data will be stored in the Ignition **Internal Database**, an embedded database that is managed by the platform, though in special cases it is necessary to store data directly to disk.

**The Internal Database**

Nearly all configuration data is stored in the internal database. This database is automatically replicated through the redundancy system, and provides a host of benefits over traditional file based storage. Furthermore, there are several helpful abstractions build on this system that make it very easy to perform common tasks like store and retrieve settings, project resources, etc. Ultimately, it is very unlikely that you will ever interact directly with the internal database though a SQL connection.

**The PersistentRecord ORM system**

Ignition includes a simple, but highly functional, ORM (object-relation-management) system, making it very easy to store and retrieve data. This system manages the creation and maintenance of tables, while still providing a high level of advanced accessibility. The PersistentRecord system is most often used directly for storing module configuration data and other information that is global to the server, that is, not part of a project.

**The ProjectResource system**

Project data is handled through the project management system (ProjectManager in the gateway
context), which in turn uses the PersistentRecord system mentioned above. The project management system makes it trivial for a module to store and retrieve any type of data required for operation, as long as it makes sense that the data should be local to a project. It's important to remember that a system may have multiple projects running at once, and a project might be cloned and run multiple times. The project management system also provides facilities for resource change history, multi-staged resource lifecycles (ie, "staged" vs. "published" resources), and the option to protect resources from modification once deployed (the "runtime lock" feature).

**Disk Based Storage**

It is occasionally necessary to store data directly to the gateway's hard drive. This is possible, and files and folders can be created under the gateway directory, but it's important to realize that the data in these files will not be included in project exports, database backups, or transferred to redundant nodes. Many times, however, it is these traits exactly that make storing directly to disk attractive.

Currently there are only a few pieces of Ignition that store data in this way: the redundancy settings, the alert state cache, and several drivers' tag data caches.

### 2.8.4 Extension Points

The Extension Point system is a single, unified interface for extending various parts of the Ignition platform. Using extension points, modules can provide new implementations of various abstract concepts. The Extension Point system is closely tied to the persistence and web interface system, reducing the amount of work required to expose and store configuration data.

**Current Extension Point Types**

The following systems expose themselves as extension points, meaning that modules can provide new implementations of them:

- Alert Notification
- Alert Logging
- Audit Profiles
- Authentication Profiles
- OPC Server Connections
- SQLTag Providers

To get started building an extension point implementation, see the Extending Ignition with Extension Points chapter.

### 2.8.5 Programming for Redundancy

Ignition supports redundancy, in which two gateways share configuration, and one is "active" at a given time. Obviously, it is crucial that module developers consider early on in the development process how this affects their module. There are two main aspects that must be examined: configuration, and runtime.

Generally, only modules that operate in the gateway scope need to be concerned with redundancy. Client and Designer modules will usually not need to know the state of the gateway their connected to.

The specific aspects of dealing with redundancy will be highlighted as they arise in other sections. This is only intended to lay out the scope of what you, the developer, must keep in mind when designing a module.
Configuration

For redundancy to work correctly, it is crucial that the two gateway nodes have the same information in their configuration databases. Configuration is synchronized from the master to the backup node as quickly as possible, in the form of incremental updates. If an error occurs or a mis-match is detected, a full gateway backup is sent from the master.

Anything that a module stores in the internal database must be sent across to the backup node. Conveniently, when using the PersistentRecord system, this is handled automatically behind the scenes. Other changes can be duplicated to the backup through the RedundancyManager in the gateway. Using that system, any operation that would change the internal database is executed as a runnable, and on success, is sent across and executed on the backup.

Runtime

The runtime considerations for redundancy come in two forms: runtime operation, and runtime state. The first, operation, is how the module acts according to the current redundancy state. There are several aspects of redundancy that must be address, such as the meaning between "cold, warm and active", and how historical data is treated. Your module can subscribe to updates about the current redundant state, and respond accordingly.

The second category of concern is runtime state. This is the "current operating state" of your module, and is the information that the module would need to begin running at the same level on the backup node, if failover should occur. In many cases, modules can just start up again and recreate this, but if not, it is possible to register handlers to send and receive this runtime state data across the redundant network. For example, in the alerting system in Ignition, alert messages are sent through the runtime state system, so that on failover the current states, including acknowledgements, is accurate. In this case, if the data was just recreated, it would not have the acknowledgement information, and would likely result in new notifications being sent.

2.8.6 Localization

Localization is the process by which an application can be adapted to use a different language, and to present information in a way that is consistent with what users in different countries are accustomed to (date formatting, for instance). Ignition supports localization, and it is fairly easy for module developers to adapt their code to support it as well, though it is easier if the understand how the system works from the outset.

At the core of localization is the idea of externalizing string data. That is, any time you would have a string of text in English, instead of using the string directly, you store it externally, and reference it through the localization system. In Ignition, these strings are stored in key/value properties files, called resource bundles. Beyond externalizing strings, in presenting data it is important to take care that numbers and dates are converted to strings using a locale aware mechanism instead of direct toStrings.

BundleUtil - The heart of externalization in Ignition

Nearly all operations involving localized string data in Ignition go through the statically access BundleUtil class. Modules can register resource bundles through a number of convenient functions on this class, and can retrieve the value using the resource key (sometimes also referred to as the "bundle key").

For example, it is common that a gateway module will have one main resource bundle defining most of
its strings. If the file were located directly next to the gateway hook class, and were named
"module_gateway.properties", in the startup function of the module it could be registered as follows:

```
BundleUtil.get().addBundle("modgw", this.getClass(), "module_gateway");
```

This registers the bundle under the name "modgw". Anywhere in our gateway module that we wanted to
display text defined in the file, for example a resource key "State.Running" that corresponds to "State is
running", we could do:

```
BundleUtil.get().getString("modgw.State.Running");
```

There are a variety of overloads for loading bundles and getting strings. See the BundleUtil JavaDocs for
more information.

Other Localization Mechanisms

Some systems support other localization mechanisms. For example, in developing gateway web pages,
placing a properties file next to a Java class with the same name will be enough to register that bundle
with the system. Mechanisms such as these will be described in the documentation as they come up.

Localization and Platform Structures

Many parts of the system that appear to use strings actually require a resource key instead. That is,
when implementing a function defined by a platform interface that returns a string, take special care to
identify whether the value returned should be the actual value, or a resource key that can be used to
retrieve the value. This should be noted in the documentation of the function, but most functions and
arguments use naming conventions such as "getTextKey" and "function(descKey)" to indicate this.

Translating Your Resources

Resources are loaded based on the user's current locale, falling back to the best possible alternative
when the locale isn't present. The Java documentation for the ResourceBundle class explains the
process. To provided translations for different locales, you can simply place the translated files (properly
named with the locale id) next to the base bundles.

To aid with the translation process, Inductive Automation is developing a translation toolkit and
localization portal that will be available to module developers by the 3rd quarter of 2011.

2.8.7 Style and Naming Conventions

In general, the Ignition platform follows the recommendations of the Oracle Code Conventions for the
Java Programming Language (see: http://www.oracle.com/technetwork/java/codeconv-138413.html).

Most of the interfaces and class in the Ignition platform are simply named using the standard casing.
Some items, however, have names that are products of their history, which may make them a bit
confusing. The following list tries to identify inconsistent or legacy naming schemes that module writers
are likely to encounter:

- `factorysql` or `factorypmi` packages, or the abbreviations "fsql" and "fpmi" in identifiers. These
  products were the predecessors to the SQL Bridge and Vision modules respectively, and these
  names still show up in code fairly regularly.
- "SR*" naming convention. During main development, Ignition was referred to as "ScadaRail". This led
to many classes being named with the initial abbreviation SR, a practice that has been abandoned,
but not completely reversed.
Programming for the Gateway

Part III
3 Programming for the Gateway

3.1 Getting Started - The GatewayModuleHook

The first step to creating a gateway scoped module is to create a GatewayModuleHook. That interface defines all of the functions that Ignition expects a module entry point to have. Almost always, though, you'll want to simply extend from AbstractGatewayModuleHook, which provides empty implementations of the functions, allowing you to implement only what you need.

Life-cycle Functions

When you extend from AbstractGatewayModuleHook, there are a few life cycle functions that must be implemented:

setup(GatewayContext context)
  Gives the module a chance to register objects with the platform, and prepare to run.
startup(LicenseState license)
  Starts execution of the module.
shutdown()
  Called when the platform is shutting down, or the module is being removed/restarted.

You'll notice that the setup function is provided a GatewayContext. This is the access point for the module to the rest of the system, and should be stored by the module. Most of the subsequent sections describe operations that would be performed by the module via the GatewayContext. In several cases, however, the module provides information to the platform by overriding functions in the module hook. For this reason, it is useful to view the JavaDocs for GatewayModuleHook and acquaint yourself with the methods defined there. These methods will also be covered in the next few sections.

3.2 Providing Status Information

Most modules that operate in the gateway scope are long running, and therefore should provide easy access to status information. There are a variety of ways to provide status in Ignition, and the requirements of your module will dictate which, or which combination, you use.

Gateway Homepage Panels

The home pages panels are the sections display when the Ignition gateway is first accessed. These panels usually indicate to the end user that a particular module is up and running, and give some basic information that should be useful at a glance. For example, the SQLBridge module shows the number of running, errored, and stopped groups—information that is fundamental for that module.

A module can provide any number of homepage panels. The panels are defined in the module's GatewayModuleHook, by implementing the getHomePagePanels() function. That function returns a HomepagePanelDescriptor, which defines various properties of the panel, such as the name, title, "help" bubble text, and more. The core body of the panel is generated by the call to newPanel on the descriptor, which returns a Wicket web panel.

What is Wicket?

Wicket is a Java web development framework, and is the framework that the Ignition gateway is built in. It provides a way to model web components as Java objects, and separate programming concerns
from display concerns. In many cases, Ignition takes care of building wicket components for you, but occasionally it is necessary or desirable to make your own. In these cases, it is most common to create Panels, which are fragments of pages. The example projects show how to create several different panels.

For more information about Wicket, see the project home page at http://wicket.apache.org/

**Status Panels**

The status panels are similar to the homepage panels in how they're created and delivered, but are shown on the status page. They are suitable for showing in-depth information that would be appropriate for troubleshooting and performance monitoring. They are implemented through the `INamedTab` interface, though normally you would just extend the `AbstractNamedTab` class.

**System Map Components**

The system map is a visual representation of the full system. It allows users to see at a glance the overall state, and is shared with redundant nodes in order to provide a complete overview.

To add components to the system map, the module should implement the `updateSystemMap(SystemMap)` function in the `GatewayModuleHook`. This function is called each time the map should be updated, and the module should create a `SystemMapElement` and add it to the map provided to the function. Actual status information is added in the form of "StatusItems" on the `SystemMapElement`. In other words, a `SystemMap` consists of multiple `SystemMapElements`, each with multiple `StatusItems`.

3.3 **Protecting Modules with the Licensing System**

The Ignition platform allows a single license file to be installed at a time. This license file contains information about all of the "licensed" modules, such as whether they've been purchased, along with any restrictions that might be included in them.

When designing your module, you will need to decide whether it will be *free*, *freely licensed*, or *commercial*. There is a core difference between *free* and the other two options in that modules marked as "free" in their module hooks do not use the licensing system. Other modules, whether they are cost a lot or nothing, must be licensed.

**How licensing works**

As mentioned above, the heart of the licensing system is the license file that gets installed during the process of *activation*. The license file is encrypted by Inductive Automation, based on a system identifier provided by the target machine during the first step of activation. Once the license file is generated and installed, only the machine it's installed on can read it.

The license management is handled by the `LicenseManager`, which can be consulted at any time to retrieve the `LicenseState` of a module. The license state is provided to a module on startup, and modules are notified of changes to the state through the `notifyLicenseStateChanged()` function defined in the `GatewayModuleHook`.

**Working with the demo system**

The Ignition platform provides a demo system that dictates 2-hour periods that can be reset by the user. Non-free modules should respect this demo, and stop functioning when the demo expires. This can easily be done by watching the `LicenseState` and observing two properties:
1) LicenseState.getLicenseMode()==LicenseMode.Trial
2) LicenseState.isTrialExpired()

When a module is activated, the license state provided to it will be updated accordingly, and the
getLicenseMode() function will return LicenseMode.Activated instead. When the demo is reset, the
isTrialExpired() function will return false. Any change in the properties will cause the
notifyLicenseStateChanged() function to be called in the module hook.

Working with LicenseDetails

Ignition allows much finer grain control over licenses beyond being simply "activated or not", in the form
of LicenseDetails. This object, retrievable through the LicenseState, exists on the platform level as
well as for individual modules, and specifies the version that the license is good for, and more
importantly, the LicenseRestrictions defined for it.

LicenseRestrictions are simply name/value pairs that can be embedded in the license file and used to
define different levels of functionality. For example, the SQL Bridge module has a LicenseRestriction
called "edition", which can currently be set to "standard" or "historical". Other license restrictions might
define the number of tags that can be used by the module, or the number of components instantiated.
This free form mechanism gives module writers full control over what different editions of their module can
do, simply by changing the license that gets sent during activation.

3.4 Storing data with PersistentRecords

The PersistentRecord system allows you to easily store and retrieve data to and from the internal
database system. As described in the design overview Storing Configuration Data, it is advantageous in
that you do not need to worry about replicating data to redundant nodes, and you do not need to worry
about writing raw SQL queries, or maintaining tables. Additionally, the PersistentRecord system offers
convenient methods for listening for changes to records.

When to (and not to) use PersistentRecords

The persistent record system is used to store any data that should be replicated through redundancy,
and offers facilities for working both with single rows (such as module settings) and many rows (such as
associated objects). It should not be used directly for project data, though, as the ProjectResource
system offers richer facilities for that.

Using PersistentRecords - Overview

The process of using persistent records is as follows:
1) Define the data by creating a class that extends from PersistentRecord, and defines fields.
2) Register the record and verify the table by calling GatewayContext.getSchemaUpdater().
updatePersistentRecords
3) Use GatewayContext.getPersistenceInterface() to create records, query, and save data.
4) Listen for changes to your record, if desired, by adding an IRecordListener to its META object.

Defining a PersistentRecord

A custom PersistentRecord type starts by extending that class. Then, it's simply a matter of defining a
series of public, static field variables, and a public "META" object.

The RecordMeta object
RecordMeta is used to create new instances, identify records, and more. Given the frequency with which it's used, it is commonly declared as a public static field in your custom record.

Field Definitions
Fields are defined using special classes that represent different data types, and are declared as public static variables. The full list of data types can be found in the JavaDocs, but some of the most common are: IntField, DoubleField, DateField, StringField, BooleanField, EnumField. The fields are defined with a name, and support various operations, such as setting a default. Additional aspects of the field are defined by the SFieldFlags, such as whether the field is a key. See the JavaDocs for that enum for more information.

Identity Fields
Another important field type is the IdentityField. This field translates to a unique, automatically assigned long value. It also defines itself as the primary key, or identifier, for the record. In most cases, it's convenient to use the IdentityField as the unique element in a record, however, it's possible to declare any field as the primary key using the SFieldFlags.SPRIMARY_KEY field flag.

References to Related Records
Often records will be related to other records, and by defining that relation, it's easy to navigate them when querying. To define a relationship to another record, two field definitions are required: a field that represents the primary key of the other record, and a ReferenceField that uses the first field to connect the two.

For example, in internal SQLTags, there are TagRecords, and TagPropertyRecords. Each tag can have many properties. The TagPropertyRecord relates back to the tag record using the tag's Long id. This is in the TagPropertyRecord:

```
public static final LongField TagId = new LongField(META, "TagId", SFieldFlags.SPRIMARY_KEY);
public static final ReferenceField<TagRecord> Tag = new ReferenceField<TagRecord>(META, TagRecord.META, "Tag", TagId);
```

Setting and Retrieving Values of Fields
Each field type has associated get and set functions defined for it in the PersistentRecord base class. For example, to get the value of a string field called "Name", you would use getString(Name). Often, for code clarity, it is better to define standard getters and setters in the record that delegate to these functions. The example below illustrates this technique.

Example
```
public class MySettingsRecord extends PersistentRecord {
    public static final RecordMeta<MySettingsRecord> META =
        new RecordMeta<MySettingsRecord>(MySettingsRecord.class, "MySettings")
            .setRecordClass(MySettingsRecord.class)
            .setSRecordClass(MySettingsRecord.class);  

    public static final IdentityField Id = new IdentityField(META);
    public static final StringField Name =
        new StringField(META, "Name", SFieldFlags.SMANDATORY, SFieldFlags.SDESCRIPTIVE);
    public static final BooleanField TurboEnabled =
        new BooleanField(META, "TurboEnabled", SFieldFlags.SDESCRIPTIVE).setDefault(true);
    public static final IntField CommunicationTimeout =
        new IntField(META, "CommunicationTimeout").setDefault(5000);

    public Long getId(){
        return getLong(Id);
    }

    public String getName(){
        return getString(Name);
    }
}
```
Registering the PersistentRecord

In order to store and retrieve your persistent record, it must be “registered” with the persistence system, so that the table can be created. This is done using the SchemaUpdater provided by the gateway context, and is normally done in the startup function of the module or class that uses the record:

```java
public void setup(GatewayContext context) {
    context.getSchemaUpdater().updatePersistentRecords(MySettingsRecord.META);
}
```

The SchemaUpdater has additional functions for ensuring that only a single instance of a record exists, which is useful for defining a single settings record instance.

Creating and Saving Record Instances

All querying, creating, and updating operations for a PersistentRecord are executed through the PersistenceInterface. The JavaDocs for that interface outline all of the available functions, but to create a basic record and save it you can use the createNew() and save() functions. For example:

```java
MySettingsRecord r = context.getPersistenceInterface().createNew(MySettingsRecord.META);
r.setName("Test");
r.setCommunicationTimeout(1000);
context.getPersistenceInterface().save(r);
```

Any time that data is modified in a record it is not stored to the database until save() is called. Note that there are other, more advanced, methods for saving records, but this convenience function is used most frequently for single records.

Querying Records

Retrieving records is also done through the PersistenceInterface, and can be accomplished with the query() and find() functions. The find function differs in that it takes primary key values and finds a specific tag. The query function allows you to build more complex search queries, and specify additional properties such as ordering.

Queries are built using the SQuery object. This object supports chained invocation, meaning that each function on it returns the object again, so properties can be chained, such as new SQuery().eq(field1, value).notnull(field2);

Example - Find all instances of MySettingsRecord with turbo enabled

```java
SQuery<MySettingsRecord> query =
    new SQuery<MySettingsRecord>(MySettingsRecord.META).isTrue(MySettingsRecord.TurboEnabled);
List<MySettingsRecord> results;
results = this.getGatewayContext().getPersistenceInterface().query(query);
```
Aggregating Operations using Sessions

When performing any operation, the persistence system behind the scenes uses sessions and datasets to execute the commands. In the examples above, sessions are created automatically, and are closed once the data is modified or delivered. When performing multiple operations together, for performance and data consistency, it is often useful to create your own session on purpose, modify it's dataset, and then commit it when finished.

A PersistenceSession object is generated by calling getSession() on the persistence interface. After retrieving a session, you can get its dataset with getDataSet(). The dataset can be used to modify rows, and then when finish, committed through session.commit(). If an error occurs, all changes can be reversed with session.rollback().

Listening for Changes to Records

Often, you'll want to be notified when a type of record that you're interested in changes. This can easily be accomplished by implementing an IRecordListener and registering it on a record's META. This interface defines functions that will be called any time a record of the specified type is added, removed, or modified. If you're only interested in one or two of these operations, it is easier to simply extend the RecordListenerAdapter class and override the functions you want.

For example, if we wanted to be notified any time our MySettingsRecord was modified, we could do something like this in our module setup function:

```java
MySettingsRecord.META.addRecordListener(new RecordListenerAdapter<MySettingsRecord>() {
    @Override
    public void recordUpdated(MySettingsRecord record) {
        applyNewSettings(record);
    }
});
```

In this example block, we're using the ability to declare an anonymous subclass inline in order to extend RecordListenerAdapter and provide our own implementation of recordUpdated. When it's called, we simply call a different function that should update our module with the new settings.

3.5 Projects and Project Resources

Beyond system wide configuration data, the most common data used by modules is project-based resource data. The project management system in Ignition provides a variety of facilities to help modules store and retrieve data. The actual information stored is opaque binary data, allowing the module to use any scheme that is appropriate.

Project resources are usually created through the designer, so this section will focus on how the gateway accesses and uses the resources. More information about projects and project resources can be found in the Design Concepts section and in the Designer chapter.

Identifying Project Resources

Project resources are identified by their resource type, module id, and scope. Additionally, each resource has a numeric id, and a universally unique id. When a module defines a new project resource, it will define the resource type, which is a free-form string. It will also define the scope of the resource. Resources scoped to the gateway, for example, will not be sent to clients. The project management system makes it easy to retrieve resources for a given module or of a given type, for the specified scope.
Loading Project Resources

Modules in the gateway can retrieve resources through the ProjectManager provided by the GatewayContext. The first step is to retrieve the Project, and then from there retrieve the resources, based on type, module, or other parameter. After retrieving the resource, the data can be extracted and deserialized.

Retrieving Projects

It's important to realize that the Project class serves a variety of purposes. It can be a fully loaded project, but can also represent a subset of a project, or simply the structure, without resource data actually present. Therefore, the first in loading data it to make sure that you are using the correct project object.

Often modules will start by calling ProjectManager.getProjectsFull(). This will return a list of all projects, fully loaded. Since Ignition supports any number of projects, it's important that modules are designed to handle them as well. It is usually advisable to define a "project runner" object that runs each project in a sandbox, as interaction between projects is rarely expected. If the id or name of the project is known, it can be retrieved directly with the ProjectManager.getProject() functions.

When retrieving projects, a ProjectVersion is specified. Each project potentially exists in two states—Staging, and Published. Many times there is little difference, as projects are set to "auto-publish" by default. However, it is important to make a distinction early on between these two versions, and decide how your module should treat them. If you want to update the running state each time a change is made and the project saved, load the Staging version. If you only want to update the state when the user "publishes" the project, only load the Published version. Since project resources always go to staging first, if the action of publishing doesn't mean anything to your module, it is not necessary to load both versions.

Loading Resources

Once a project is retrieved, resources can be retrieved through several functions on the Project object. Most commonly, modules will want to load specific types of resources at a time, which can be accomplished with the Project.getResourcesOfType() function.

Deserializing Resource Data

Once the project resource is retrieved, the data can be accessed through ProjectResource.getData(). If the data was serialized through Ignition's XML serialization mechanism, the full code to deserialize a resource object might look like:

```java
MyModuleResource obj = (MyModuleResource) XMLDeserializer.deserialize(gatewayContext.createDeserializer(), resource.getData(), log);
```

Important: When deserializing, it is important to use the deserializer created by GatewayContext.createDeserializer(). Do not create your own XMLDeserializer.

Listening for Project Changes

Modules can add a ProjectListener to the ProjectManager in order to be notified when a project is added, removed, or modified. It is important to remember that the listener will be notified for all...
projects, and that the Project object provided will only be the difference between the last project state and the new one. If the full project is required, for example to resolve full folder paths of resources, you should retrieve it from the ProjectManager using the id provided by the supplied project.

3.6 Extending Ignition with Extension Points

As described in the Extension Points section of the design overview, extension points are hooks that allow modules to implement new versions of abstract ideas, such as a new type of authentication profile, or a new alert logging manager.

How Extension Points Work

Various parts of the system have been defined as "extension points". Each one of these parts has an extension point type defined for it, and a manager that handles the book keeping of registered types. There is also a base settings record defined for the type.

When the system starts up, modules register new extension point implementations with the managers. When a user chooses to create a new instance of something, such as an authentication profile, the system looks at all of the registered types, and displays them to the user. When the user selects one, a new base settings record is created for the new "profile". If the extension point implementation defines additional settings, an instance of that persistent record will also be created and linked to the base record.

When it comes time to start up the profiles, the manager will locate the registered type and call its create function, providing it with the GatewayContext and the base profile setting record. The custom type can use this profile record to load its settings record, and instantiate the implementing class.

Defining a new Extension Point

At a high level, the process for creating a new extension point implementation is as follows:

1) Create a class that implements the desired type of object
2) Define the extension point type
3) Define any settings that will be required for the implementation
4) Register the extension point with the appropriate manager.

Implement the Desired Object

Ultimately, the goal is to provide your own implementation of some piece of the system. At the heart of this, of course, is your implementation. The rest of the Extension Point system is for book keeping and linking in your class. The table below specifies which interface to implement for each extension point.

Define the Extension Point Type

Each system in Ignition that exposes an extension point will define an abstract implementation of ExtensionPointType. In addition to default implementations, it will also define a function to instantiate an instance of the type. Your extension point definition will implement this function in order to load and configure your implementation at runtime.

Define the Settings Record

If your implementation has its own settings, you can define your own PersistentRecord to hold them. The persistent record that you define must have a reference to the profile record for the Extension Point you are implementing.

For example, if implementing an authentication profile with custom settings, your settings record would have the following reference in it:

public static final LongField ProfileId = new LongField(META, "ProfileId", SFieldFlags.SPRIMARY...
public static final ReferenceField<AuthProfileRecord> Profile = new ReferenceField<AuthProfileRecord>(META, AuthProfileRecord.META, "Profile", ProfileId);

It's worth noting that in that example, the Profile field is used not only as the foreign key, but also the primary key for our settings record. Therefore, for this persistent record, there would not be a separate IdentityField defined.

Register the Extension Point
The final task is to actually register the extension point with the system, through the appropriate manager. In the `startup()` function of your `GatewayModuleHook`, use the manager specified below to add your type.

### Extension Point Object Reference

<table>
<thead>
<tr>
<th>Extension Point</th>
<th>Interface to implement</th>
<th>Base extension point type to extend from</th>
<th>Profile record to reference</th>
<th>Manager to register with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alert Notification</td>
<td>AlertNotification</td>
<td>AlertNotificationProfile</td>
<td>AlertManager</td>
<td></td>
</tr>
<tr>
<td>Alert Storage Profile</td>
<td>AlertStorageProfile</td>
<td>AlertStorageProfileRecord</td>
<td>AlertManager</td>
<td></td>
</tr>
<tr>
<td>Audit Log AuditProfile</td>
<td>AuditProfileType</td>
<td>AuditProfileRecord</td>
<td>AuditManager</td>
<td></td>
</tr>
<tr>
<td>Authentication AuthenticationProfile</td>
<td>AuthenticationProfileType</td>
<td>AuthenticationProfileRecord</td>
<td>AuthenticationManager</td>
<td></td>
</tr>
<tr>
<td>OPC Server SROPCServer</td>
<td>OPCServerType</td>
<td>OPCServerSettingsRecord</td>
<td>OPCManager</td>
<td></td>
</tr>
<tr>
<td>Realtime SQLTagProvider</td>
<td>SQLTagProviderType</td>
<td>SQLTagProviderRecord</td>
<td>SQLTagManager</td>
<td></td>
</tr>
</tbody>
</table>

### 3.7 Storing History using Store and Forward

The store and forward system works as a pipeline to deliver data reliably to the database. The database is a data sink, and data is buffered in the store and forward engine using a mixture of memory and disk storage, until it can be delivered. If the data results in an error, it is quarantined in order to allow other data to pass. While it is buffered, it may be combined with similar data, in order to improve efficiency by writing in bulk transactions instead of individual units.

Modules have the ability to feed data into the store and forward system, either by using existing framework classes or by defining new types of data whose storage functions can be customized.

### The Core Pieces

At its heart, a store and forward pipeline consists of a DataSink and HistoricalData. The data sink receives the historical data. Both of these interfaces are extended and implemented by numerous pieces that operate in different manners, but fundamentally the historical data is generated, and passed through a series of sinks. Since a sink can only receive data, most sinks are extended to be a DataStore as well. DataStores are sinks, but can also provide back the data in the form of TransactionSets, which are simply lists of ForwardTransactions. A ForwardTransaction is a wrapper around a piece of HistoricalData, with functions for reporting success or failure.

### Accessing the Store and Forward System

The store and forward system is managed by the HistoryManager, provided by the
The history manager is used to put data in the system for a particular pipeline, as well as to register new data sinks.

### Storing Data

There are currently two types of data flavors that are supported by the default store and forward engines: tag history data, and datasource data. The tag history data, clearly, is used to store SQLTags History data, and defined by the HistoricalTagValue and implemented by BasicHistoricalTagValue, and the slightly more efficient PackedHistoricalTagValue. To store history for tags, you can use either of these classes and send them to the history manager for the datasource you want to store to.

More commonly, however, modules will want to store general database data. The DatasourceData interface provides a flexible method for storing arbitrary data. After passing through the store and forward system, `storeToConnection()` is called with an open connection. The implementation of the data can then do anything that it needs to do to store the data.

If you simply want to store a basic row of data in an existing table, you can save yourself some work and use the BasicHistoricalRecord class. It allows you to define columns, and then add rows of data. It supports multiple rows of data at once, stores efficiently in the data cache, and automatically groups with like data to form efficient transactions. The columns are defined by the HistoricalColumn interface, and there are two general purpose implementations, the `ValueHistoricalColumn` for most values, and the `LiteralHistoricalColumn` for including a keyword in the query.

### 3.8 Working with Platform Services

The Ignition Platform offers a wide range of services that modules can build on, instead of implementing themselves. The services presented in this chapter are provided through the GatewayContext given to each module.

#### 3.8.1 Databases

Database access is at the core of Ignition, and is used by many parts of the system. The platform manages the definition of connections, and provides enhanced classes that make it easy to accomplish many database tasks with minimal work. When necessary, full access is also available to JDBC connections. Database connection pooling is handled through the Apache DBCP system, so module writers do not need to worry about the efficiency of opening connections (though on the opposite side, it's crucial that connections are properly closed). Additional features, such as automatic connection failover, are also handled by the platform.

#### Creating a Connection and Executing Basic Queries

All database operations are handled through the `DatasourceManager` provided by the `GatewayContext`. The `DatasourceManager` allows you to get a list of all of the defined datasources, and then to open a new connection on them. The returned connection is an `SRConnection`, which is a subclass of the standard JDBC Connection object that provides a variety of time saving convenience functions. Remember, though, that even when using the convenience functions the connection must still be closed.

**Example:**

```java
SRConnection con;
int maxVal;
try{
    con = context.getDatasourceManager().getConnection("myDatasource");
```
Note that this example does not handle potential errors thrown during query execution, but does illustrate the best practice of closing the connection in a Finally block.

The SRConnection class also provides the following useful functions:
- `getCurrentDatabaseTime()` - Shortcut to query the current time.
- `getParentDatasource()` - Provides access to the datasource object that created the connection, which can provide state information and access to other important classes, such as the database translator.
- `runPrep*()` - Several functions that send values to the database through prepared statements.

Prepared statements, such as the one used in the example above, are generally preferred to text queries as they are safer and less prone to errors.

**Executing Complex Transactions**

The SRConnection extends from the standard JDBC Connection object and can be used in the same way. This means it is possible to run multi-statement transactions with rollback support, and to use batching for high-performance data insertion. For more information, consult any JDBC guide.

**Verifying Table Structure**

When creating database-centric modules, it is very common to expect a table to exist, or to need to create a table. Ignition provides a helper class called `DBTableSchema` that can help with this task. The class is instantiated with the table name and database translator to use (provided by the datasource). Columns are then defined, and finally the state is checked against the given connection. Missing columns can be added, or the table created if necessary.

For example, the following is a common way to define and check a table, creating it if required:

```java
SRConnection con;
DBTableSchema table;
try{
    con = context.getDatasourceManager().getConnection("myDatasource");
    table = new DBTableSchema("example", con.getParentDatasource().getTranslator());
    table.addRequiredColumn("id", DataType.Int4, EnumSet.of(ColumnProperty.AutoIncrement, ColumnProperty.PrimaryKey));
    table.addRequiredColumn("col", DataType.Int8, null);
    table.verifyAndUpdate(con);
}finally{
    con.close();
}
```

**3.8.2 Execution Scheduling**

Performing some task on a timer or in a different thread is an extremely frequent requirement of gateway based modules, and the Ignition platform makes it easy by offering `ExecutionManager`. In addition to managing time-based execution, the execution manager can also execute a task once or allow the task to schedule itself, all while providing status and troubleshooting information through the gateway webpage. Private execution managers can be created in order to allocate threads for a specific task, though for most uses, the general execution manager provided by the gateway context should suffice.
Registering Executable Tasks

Anything that implements Java's Runnable interface can be registered to execute with the execution manager. Tasks can either be executed once with the `executeOnce()` functions, or can be register to run repeatedly with the various `register*()` functions. Reoccurring tasks must be registered with an owner and a name. Both are free form strings, and are used together to identify a unique unit of execution, so that it can be modified and unregistered later.

After a task is registered, it can be modified later by simply registering again with the same name. To stop the task, call `unRegister()`. Some functions in the execution manager return `ScheduledFuture` objects, which can be used to cancel execution before it happens.

SelfSchedulingRunnable Tasks

Most tasks are registered at a fixed rate and rarely change. In some cases, though, the task may wish to frequently change its rate, and re-registering each time is inefficient. In these cases, instead of supplying a Runnable, you can implement SelfSchedulingRunnable instead. After every execution, the SelfSchedulingRunnable provides the delay to wait before the next execution. When it is registered, it is provided with a `SchedulingController` that can be used to re-schedule the task at any time. For example, a self scheduling task could run every 30 seconds, and would normally return 30000 from the `getNextExecDelayMillis()` function. A special event could occur, however, and the task could be executed at 500ms for some amount of time. The self scheduling runnable would call `SchedulingController.requestReschedule()` and would then return 500 until the special event was over.

Fixed Delay vs. Fixed Rate

Executable tasks are almost always registered with fixed delays, meaning that the spacing between executions is calculated from the end of one execution to the start of another. If a task is scheduled to run every second, but takes 30 seconds to execute, there will still be a 1 second wait between each event. Some functions in the execution manager allow the opposite of this, execution at a fixed rate. In this case, the next execution is calculated off of the start of the event. If an event takes longer than the scheduled delay, the next event will occur as soon as possible after the first completes. It's worth noting that events cannot "back up". That is, if a task is scheduled at 1 second, but the first execution takes 5 seconds, it will not run 5 times immediately to make up the missed time. Instead, it will run once, and will then start to follow the schedule again.

Creating Private Execution Managers

In situations where the tasks being registered might take a long time to execute, and several of them may run at once, it is usually better to create a private execution manager. The private managers work the same as the shared manager, but do not share their threads. That way, if tasks take a long time to execute, other parts of the system won't be held up. A private execution manager can be created by calling `GatewayContext.createExecutionManager()`. When creating a manager, you must give it a name, and decide how many threads it will have access to. It is important to choose wisely, as too many threads will waste system resources, but too few might lead to thread starvation, where a task is waiting to execute, but no threads are available to service it.

3.8.3 Auditing

The Audit system provides a mechanism for tracking, or "auditing", events that occur in the system. The events are almost always associated with a particular user, in order to build a record of "who did what"
when”. Audit events are reported through an **AuditProfile**, set on a per-project level. Any module that wishes to track user actions can report audit events to the profile specified for the current project.

### Reporting Events
Adding events to the audit system is as simple as generating an **AuditRecord** and giving it to an **AuditProfile**. Instead of implementing the AuditRecord interface yourself, you'll almost certainly want to simply use the **DefaultAuditRecord** class.

With the project id or project name, the AuditProfile can be retrieved through the AuditManager provided by the GatewayContext.

### Querying Events
Modules can also access the history of audit events by using the `query()` function on the AuditProfile. This function allows you to filter on any combination of parameters in the AuditRecord.

#### 3.8.4 OPC
OPC is an industrial standard for communication that is widely used to provide access to nearly every type of device. OPC works on a client/server basis, with the server talking the device's language and translating data for the OPC client. While Ignition contains a built in OPC server, this section describes the client capabilities. There are several versions of the OPC specification, but the Ignition OPC system provides a single abstracted layer that hides the differences between them.

The OPCManager, accessible through the gateway context, provides the ability to Browse, Read, Write, and Subscribe to OPC data. Connections to OPC servers are handled by the platform and defined in the gateway, and besides the server name that is included in addresses, the module using the system does not need to be aware of the different servers.

### Identifying Addresses - The ServerNodeId
Each data point (tag) in OPC is identified by it's Server Name, and it's Item Path. The newer OPC-UA specification makes room for more complex identifiers, such as GUIDs, and namespace based organization. The **ServerNodeId** object allows for both schemes. The ServerNodeId can be retrieved either through browsing, implementing the interface yourself, or using the **BasicServerNodeId** implementation. NodeId, the core address of a tag, can either be instantiated directly, or off of a properly formed string with `NodeId.parseNodeId()`.

### Reading and Writing Values
It is possible to read and write values to tags at any time using the corresponding functions on the OPCManager. Both functions take a list of inputs, and return a corresponding list of outputs, guaranteed to be the same length. The returned objects will indicate the success of the operation, and provide values, if the operation was a read. It is not necessary to make separate calls for different servers, as the OPCManager will handle separating out the values for you.

If you wish to obtain the value of an OPC item on a regular basis, a more efficient mechanism than calling Read is available: the subscription system. By subscribing to an address, you can specify a rate at which you want to be notified of changes. Notifications only occur when the value has actually changed. This mechanism is more efficient for the OPC server because it is able to optimize the set of tags that it needs to read.
Managing Subscriptions

OPC tags are subscribed through the definition of a SubscribableNode, and a named subscription. Each SubscribableNode defines its address and subscription name, and provides callback functions to be notified of changes to value and quality for the tag. The BasicNodeSubscriptionDefinition class can be used to quickly define subscribed nodes.

Subscribing to OPC tags is a two step process:
1) Define the subscription: use OPCManager.setSubscriptionRate() to define a subscription with the given name and rate.
2) Create and subscribe SubscribableNodes: Implement the required interfaces, and use the subscription name defined above where necessary. Register them through OPCManager.subscribe()

Once the subscription is running, tags can be added and removed at will with the subscribe and unsubscribe functions. A subscription remains defined until OPCManager.cancelSubscription() is called, unless OPCManager.enableAutoCancel() has been called for it, and all nodes have been unsubscribed.

Understanding OPC Values and Qualities

Each OPC value has a value property, but also a quality property. The quality defines how trustworthy the value is, and communicates additional information about tag. For example, if a quality is DataQuality.Not_Found, the requested path is not available or correct, and so the value has no meaning. As you can see from the OPCManager interface, the subscribe function does not return errors. Instead, the status is communicated through the SubscribableNode.setQuality() function.

3.8.5 SQLTags

SQLTags is the realtime tag system in Ignition. Tags can be driven by OPC, Expressions, SQL Queries, or static values, and provide features such as scaling, alerting, and type conversion.

Understanding Tag Providers

When working with SQLTags, it is important to understand the architecture of how tag providers work together, and the different types of providers that exist. All tags exist inside of a Tag Provider, but depending on the type of provider, they may actually be driven by a different, remote provider.

There are currently two main types of providers that illustrate this: the internal provider, and the database provider. The internal provider is conceptually very simple- it is a tag provider that is only local to the system it is on, and it drives its own tags. The database provider comes in two forms, however, the regular form, and the driving form. The standard database provider cannot drive tags, it can only observe a database and make the tags in that database visible to the system. The database tags must be driven by a different entity, such as the driving provider on a different machine, or a FactorySQL installation.

Addressing SQLTags

All tags are addressed via the TagPath object. A tag path consists of several components: the tag provider, the folder path, the tag name, and optionally, the tag property. For example:

[MyProvider]/Path/To/Tag.Quality

The provider component can be left off if you are addressing a tag from inside a project, and that tag belongs to the default provider for the project.

The easiest way to generate TagPath objects is to use the TagPathParser class, though it is possible to construct them manually using the BasicTagPath, or implementing the interface yourself.
Subscribing to Tags
From any context, you can subscribe to SQLTags via the TagSubscriptionManager. The TagChangeListener provided to the subscribe function can specify a specific tag property to listen for, or can listen for all tag changes. When listening for all changes, it will also be notified when the tag configuration changes.

Writing to Tags
To write to a tag, you must use the TagManager provided by the context that you are in, as the write procedure differs based on scope.

3.8.6 Alerting
The alerting system works on a "bus" principal. Messages are posted to the alert bus, and are delivered to all listeners registered on that bus.

Types of Messages
All alert messages implement the AlertMessage interface. The default implementation, DefaultAlertMessage, is available for use as a simple structure that holds all of the alert variables.

A key field on the alert message is the Flags field. This value indicates the purpose of the message, and can be a combination of any of the values: Active, Cleared, Acknowledged, Register and Deregister. Most of the other fields, while important, are only relevant in regards to the flag field. For example, it is unlikely that an acknowledge message would have the "message" field defined, and a non-acknowledge active message would not have a populated AckUser.

As mentioned, virtually all combinations of the flags are possible and occur (with the exception, of course, of Register and De-register together). For example, it's possible to see Active and Cleared together, on an "any-change" state. Active & Acknowledged will occur when the "ack mode" alert setting is set to "unused", and so on.

Register and De-register Events
When a tag is first initialized, it will send its first alert event with the "register" flag set. This indicates to the listeners that the tag has in fact just been initialized, and that the alert may not be a new event. For example, imagine a system with active alerts that is restarted. On startup, the tags will be initialized and will again be in an active state. New alert messages will be generated, since some tags may have changed state while the system was offline, but will be sent with the "register" flag high. The alert notification system then may choose to not send new messages, based on other information that it has stored.

De-register events are sent only on a tag's deletion in order to instruct listeners to remove the tag from any caches.

Posting Alerts
Alerts can be posted to the bus by calling GatewayContext.getAlertBus().publishAlert() with a properly constructed AlertMessage. As mention above, most modules will simply choose to use
the `DefaultAlertMessage` class.

**Listening to Alerts**

To listen to alert events on the AlertBus, you simply need to implement the `AlertMessageHandler` interface and register it with the `AlertBus.addAlertMessageHandler()` function. The `AlertMessageHandler` interface only defines a single `receiveAlert()` function that will be called for each alert message posted.

**Implementing a Notification Profile or Storage Profile with Extension Points**

Generally most modules implementing an `AlertMessageHandler` will do so in order to provide the functions defined by a notification or storage profile. When this is the case, instead of implementing the `AlertMessageHandler` directly, they should use Extension Points to add a new profile. Doing this will improve the lifecycle management of the new profile, and will tie its configuration in seamlessly with the gateway.
Programming for the Designer

Part IV
4 Programming for the Designer

4.1 Working with Project Resources

Project Resources, as described in the Programming for the Gateway section, are the primary data persistence mechanism for modules that need to store data on a per-project basis. The definition of what a resource represents is completely undefined, allowing modules to define them in any way that they like. There are certain paradigms that generally should be followed, such as displaying the resources in the Project Resource Tree in the designer and adding actions for creating new instances to the Designer menus, but all of these off a high degree of flexibility.

Project resources, as described in the gateway and design concepts sections, are essentially raw data with a unique id. There are a variety of useful accessories built in, such as support for folder hierarchies, but at their core they are very simple. They are contained in a Project object, which may represent either the full project, or a subset of it. For example, when resource changes are communicated to the modules, an instance of Project containing only the modified rows is delivered.

Creating New Resources

The process of creating a new resource involves generating an id for it, creating the resource, and then updating the project to include it.

Step 1 - Create a new resource id

Resource ids must be created atomically through the resource lock manager. For example:

```java
long newId = designerContext.getLockManager().newResourceId();
```

Step 2 - Create new ProjectResource

Instantiate a new ProjectResource class with the new id, your module id, resource type id, and other data. For example:

```java
ProjectResource resource = new ProjectResource(newId, moduleId, resourceType, resourceName,
ApplicationScope.GATEWAY, resourceData);
```

Remember that project resources dictate a scope that indicates where they will be used. This is primarily to avoid sending resources that are only used in the gateway to the clients. The designer, of course, will have access to all resources. This example expects that you have all of the fields already defined in variables. The resource data is a byte array, and can be obtained by any method that you desire. Most commonly though, modules will simply use the serialization facilities provided by Ignition. For example, the following code could have easily proceeded the previous example:

```java
MyModuleObject newObj = new MyModuleObject();
XMLSerializer s = designerContext.createSerializer();
byte[] resourceData;
resourceData = s.serializeAndGZip();
```

Step 3 - Updating the Project

Finally, the resource is added to the project:

```java
designerContext.updateResource(resource);
```

Creating Folders

While folders are simply project resources like the others, the way that they are created is a bit different than other resource types. The procedure resembles the steps above, in that a new resource id must be generated, and the folder is defined with a scope and module id, but instead of instantiating a new ProjectResource and adding it to the project, you simply call DesignerContext.addFolder(). For example:

```java
designerContext.addFolder(newId, moduleId, ApplicationScope.GATEWAY, folderName, parent);
```
This function handles generating a new UUID for you, and also ensures that the folder name will be unique for the module.

### Modifying Resources

The general process of modifying resources is similar to that of creating them, in terms of serializing data, setting it, and calling `updateResource()`, with the exception that care should be given to managing Resource Locks.

#### What Are Resource Locks

Since Ignition supports concurrent designer sessions, it is possible that two people may try to edit the same resources at once. Allowing both parties to edit a resource and then simply using one would not be intuitive to the user. To prevent this, resources can be locked for editing, which is communicated through the gateway so that all other designers are prevented from editing the same resource. Resource locks are tracked by session, and are automatically released when the session expires or the project saved. Still, careful management of locks by the module writer is important, as it reduces the number of times users encounter unexpected locked resource errors.

#### Obtaining, Updating and Returning Locks

All lock management is performed through the `ResourceLockManager`, which is obtained through the static `ResourceLockManagerFactory.getManager()` function.

The process for properly managing locks is as follows:

1. Call `ResourceLockManager.requestLock()` for a resource id. This function returns a boolean indicating whether the lock could be obtained. If it returns `false`, you should not proceed to edit the resource.

2. Modify the resource. Call `ResourceLockManager.updateLock()` for the resource, to notify the lock manager that the resource has, in fact, been modified. This will prevent other sessions from editing the resource until the project is saved.

3. When done, call `ResourceLockManager.releaseLock()`. This is especially important if no changes have been made to the resource (and updateLock() hasn't been called) so that other designer sessions may edit the resource.

At any time, the `ResourceManager.hasLock()` function may be called to see if the current session currently has the lock for the resource. This is important because the updateLock() and releaseLock() functions require that the caller knows that it has the lock.

#### Example

This contrived example shows the general process of modifying a resource object. The actual process would involve multiple stages of locking the resource, presenting it for modification, and later committing it.

```java
long resourceId = getCurrentResourceId(); //Would return the currently selected resource
if(ResourceLockManager.getManager().requestLock(resourceId)){
    try{
        ProjectResource resource = designerContext.getProject().getResource(resourceId);
        MyModuleObject object = getResourceFromResource(resource); //This function would deserialize the data in the resource
        ... edit the resource object ...
        resource.setData(serializeObject(object)); //User defined function serialize as described above
        ResourceLockManager.getManager().updateLock(resourceId);
    }finally{
        //Make sure to release the lock, even if an error occurs
        ResourceLockManager.getManager().releaseLock(resourceId);
    }
}else{
    //Could not obtain lock, show an error
}
```
Deleting Resources

Deleting resources is as simple as calling `DesignerContext.getProject().deleteResource()` for the resource id with the `markDeleted` flag set to `true`. Note, however, that in regards to locking, the deletion of a resource should be treated the same as a modification, and the procedure outlined above should be followed.

Listening for Project Resource Changes

Listening for project resource modifications is as easy as implementing a `ProjectChangeListener` and registering it by calling `ClientContext.addProjectChangeListener()`. The `ProjectChangeListener` receives only the subset of the project that has changed, or the specific resource modified. At any time, however, it's possible to get the full project by calling `Context.getProject()`.

Folders the Project Resource Hierarchy

As mentioned in the Creating New Resources section, `ProjectResources` support hierarchical organization with folders. Folders are simply `ProjectResources` themselves, with the important distinction that their resource data is actually a UUID (universally unique identifier). This unique id stays consistent across resource renaming, and is what is used by resources to identify their parent folder. Each resource, therefore, has a "parent uuid" field that defines the folder resource it belongs to. A null value indicates that the resource is at the root level. Folders are stored according to the module that they belong to, but are all of the same resource type, defined by `ProjectResource.FOLDER_RESOURCE_TYPE`.

To help make working with folders easier, the `Project` class has a variety of functions for browsing the structure. Additionally, the `ProjectResource` class has the `getDataAsUUID()` function that makes it simple to obtain the UUID value of a folder resource.

4.2 Building a Workspace

The UI in the center area of the Designer is called the Workspace. Any module can add new workspaces to the Designer, but only one workspace is active at a time. A workspace should be designed to allow the user to edit some project resource. Typically, they select the resource to edit by clicking in the project browser. Each project browser tree node can have a workspace ID associated with it so that when it is selected, that workspace can become active.

To add a workspace, a module must create an object that implements the `ResourceWorkspace` interface. As you can see from the JavaDocs, this interface allows the workspace to provide a big `JComponent` to fill up the center workspace area, as well as associated menus, toolbars, and floating frames that should be visible when the workspace is active. The most important function of this interface is the one that returns the workspace itself, which can be any Swing component you'd like. This allows the workspace system to present any arbitrary UI to the user for resource customization.

The `resetFrames` Function

The one function in `ResourceWorkspace` that is non-obvious is the `resetFrames` function. This function is called to initialize the workspace's view, and when the user chooses the `View > Reset Panels` menu option while the resource is active. It is this function's duty to set-up any dockable frames that should be visible while the workspace is active. Some dockable frames such as the project browser and the SQLTags browser will be visible and docked along the left side by default. Other panels will be
hidden. You should set up the states and sizes of any dockable frames that you've added in your workspace as well as any built-in frames that you care about. For example, the Vision module, which wants the OPC browsing dockable frame to be hidden and then start as a popup when made visible, implements the function like this:

```java
DockableFrame opc = dockingManager.getFrame(OPCBrowserPanel.DOCKING_KEY);
opc.setInitMode(DockContext.STATE_HIDDEN - DockContext.STATE_FLOATING);
opc.setUndockedBounds(new Rectangle(350, 250, 280, 450));
opc.setInitSide(DockContext.DOCK_SIDE_WEST);
opc.setInitIndex(1);
```

While the SQL Bridge module, which wants the OPC browsing frame to be docked as a tab with the SQLTags browser, implements it like this:

```java
DockableFrame opc = dockingManager.getFrame(OPCBrowserPanel.DOCKING_KEY);
opc.setInitMode(DockContext.STATE_FRAMEDOCKED);
opc.setInitSide(DockContext.DOCK_SIDE_WEST);
opc.setInitIndex(1);
```

To learn about how the docking system works, read the documentation from the JIDE product.

**Designable Workspace**

There is a special abstract subclass of ResourceWorkspace called the DefaultDesignableWorkspace that is intended for any workspace that needs to design something using drag-and-drop WYSIWYG style manipulation. Writing this sort of system by hand is very time-consuming, and worst of all would inevitably leave the user with an inconsistent feeling workspaces. For these reasons, if your module would benefit from drag-and-drop manipulation of visually represented components, you are encouraged to use the designable workspace system. To learn about this system, read the JavaDocs for AbstractDesignableWorkspace and DesignPanel, which do the bulk of the work.

The general concept is that AbstractDesignableWorkspace is a tabbed pane, each tab holding something that should be designed. The thing being designed is your own creation, but must be some sort of JComponent that implements DesignableContainer. Each tab will hold a DesignPanel, which is a customized JScrollPane. The view of the scroll pane will hold your component, and a glasspane-like component on top of it called the InteractionLayer. When in preview mode, the interaction layer will intercept all mouse events, and draw handles, guides, the grid, and all the other things you're used to from the Vision module. Your subclass of AbstractDesignableWorkspace must know how to create an ItemDelegate for each of its children, which must know how do do various things like set the size and location of objects contained within it.

### 4.3 Creating Menus and Toolbars

The menu and toolbar system in the Designer can include contributions from modules. Each designer hook has the opportunity to add its own menu items and toolbars, as does each workspace. The menu items and toolbars from each workspace will only be visible when that workspace is active.
Actions

Before getting into the specifics of adding toolbars and the menu merging system, it would be worthwhile to discuss actions. Actions are a Swing concept that work quite well. The general idea is that instead of coding up a menu item and a toolbar button that do the same thing, you define an action object that has a name, tooltip, icon(s), and the code of how to execute the action itself. Then you simply add that action to a menu and/or toolbar, and the menu itself or toolbar itself creates the proper item or button for that action.

For a normal action that just needs to do something when clicked on, you typically make one by creating a subclass of javax.swing.AbstractAction. You may wish to use Ignition's BaseAction class, which makes it more convenient to hook an Action's text and tooltip into our internationalization system. The javadocs on BaseAction explain how it is used.

For actions that are destined to become things like radio button menu items, checkbox menu items, or mutually exclusive toolbar toggles, you can create subclasses of StateChangeAction.

Toolbars

To add a toolbars, you implement getModuleToolbars() from your designer hook, or getToolbars() from a ResourceWorkspace implementation. Both of these functions return a List<CommandBar>. CommandBar is a class from the JIDE docking framework, a 3rd party commercial UI library that the Designer uses. You can use DesignerToolbar, a subclass of CommandBar. It has handy functions to deal with creating various kinds of toolbar buttons from actions.

Menu Merge

To add menu items, you create a collection of menu merge objects. Just like with toolbars, you add these to the system either by implementing getModuleMenu() from your designer hook, or getMenu() from a resource workspace.

The menu merge system is a way to insert menu items into the existing menu structure. It relies heavily on knowing the exact names and locations of known menus. These are stored in the WellKnownMenuConstants class. See the javadocs on JMenuMerge for details.

4.4 Supporting Search and Replace

A module can tie into the search and replace system by registering a SearchProvider implementation with the DesignerContext. The SearchProvider will be used to get information about search options, and will be used to generate potential search matches.

How the Search and Replace Process Works

The basic process is simple: the SearchProvider generates potential search results (SearchObject). The search and replace system will try to match the text of the SearchObject to the search pattern. If the object passes, it will be displayed to the user. The user can then double-click on the object, which delegates to the SearchObject.locate() function, or can try to replace the value.

The SearchObjects generated by the SearchProvider are returned through an Iterator, with the idea being that they are generated on the fly, so that only matching results need to be kept in memory. The SearchObject implementation will be unique to the provider, as it must know how to locate and replace the resource that it represents.
Refining the search: Categories and Selected Objects

The SearchProvider can use two mechanisms to refine the objects to be searched. The first, categories, are a simple way to provide a broad choice. They are displayed as check boxes under the search provider entry on the search window, and the selected objects are passed back to the provider when the search is executed.

The second method is through the use of selectable objects. With this mechanism, the search provider will be displayed with a link that will call into the search provider with a SelectedObjectsHandler. The search provider can display a dialog to select which "items" should be searched. The definition of an item is free-form, and can be as simple or complex as the provider wants. Like categories, the selected objects will be passed into the main search function when the search is executed.

Common Design Paradigms and Tools

While the implementation of the search and replace interfaces is very specific to each module, there are certain concerns and patterns that come up frequently.

SearchObject Information

SearchObjects have the ability to provide an Icon, an Owner and a Name. Some experimentation is usually necessary to find the right amount of information to provide that makes the search results intuitive and easy to identify. For example, imagine the "Text" property of a Button. Returning values of "Button" and "Text" for owner and name respectively might seem clear at first, but once multiple buttons appear on a screen, it becomes impossible to distinguish them. Including the actual component name in the "owner" string solves this problem, but can then make it difficult to decipher at a glance what the result is. Using a button image for the icon can provide the final visual clue that the user needs in order to understand quickly.

Iterator Management

While the Iterator mechanism works great for reducing the amount of memory required for searching, it can sometimes be unwieldy to deal with in anything but the simplest modules. It is not uncommon to encounter a situation where you must recursively iterate through multiple levels, potentially returning results at each level. There are several important classes and libraries available that can help:

- Google Collections - Ignition includes access to the very useful Google Collections package that includes a set of classes that can help when working with iterators. The *Iterators* class, in particular, has functions for concatenating iterators and more that are very useful.

- Ignition's SearchObjectAggregator and SearchObjectCursor classes - These incredibly useful classes can be used to overcome the difficulty mentioned above. The SearchObjectCursor is an adaptation of Iterator that can return SearchObjects or other SearchObjectCursors. The SearchObjectAggregator is an implementation of Iterator that knows how to deal with the SearchObjectCursor. In other words, if you implement SearchObjectCursor, you can return either results or new cursors to search, and the aggregator will sort it out for you.

4.5 Supporting Undo and Redo

Undo and Redo is supported through the singleton UndoManager class, and the UndoAction interface. The system allows actions to be grouped based on the context of what the user is doing, and supports both independent and "aggregate" events.
Undo Contexts

The UndoManager supports the idea of multiple contexts, meaning multiple stacks of undo/redo tasks, with the active stack being determined by the screen or action that the user is currently on. For example, the SQL Bridge and Vision modules both define a context, so that the undo/redo operations are local to each. In other words, when the user is editing a group, and clicks "undo", it won't undo the last edit to a window.

To define a context, or switch to it, simply call UndoManager.setSelectedContext(). The context object can be anything, as long as it is appropriate for the key of a hashmap. When and how often you set the setSelectedContext() function will depend on the scope of your context, but it is common to call it when resources in your module are selected, a frame activated, or any other time that might indicate a switch has been made from a different context. If you don't wish to manage the context in this way, you can also simply include it in the overloaded function to add UndoActions, UndoManager.add().

The UndoAction Interface

The UndoAction represents a unit of execution that can be performed, reversed (undo), and the done again (redo). The core functions are execute(), which is the positive action and always means "do this", and its opposite, undo(). Given the semantics of these functions, it's common to define an UndoAction, call execute(), and then if successful, add it to the UndoManager.

The UndoAction.getDescription() function provides the text that will be displayed to the user. Additionally, this value is used to group similar actions, if appropriate.

Grouping Actions

There are several different ways that actions can be grouped by the UndoManager. In the first case, with normal independent events, repeat events that occur within 700ms of each other will be grouped together, and executed together when Undo or Redo are called. This provides the behavior expected by the user when performing a small event many times within a short period, such as typing, or moving a component.

The second type of grouping that is available is dictated by the UndoAction. isGroupSequenceIndependent() function. When that function returns true, only the first registered action will be called, regardless of how many times the same action was registered.

4.6 Designer to Gateway Communication (RPC)

When building a module with both Designer and Gateway components, it is often necessary to provide some sort of communication link between them. The project management system handles notifying the gateway of resource changes, but for more dynamic communication, the RPC (remote procedure call) system is available.

The RPC system is easy to use, and works as follows:
1) In the designer, the module uses GatewayConnectionManager.getInstance().getGatewayInterface().moduleInvoke(...) to send an invocation call to the gateway for the provided module id, with the given function name and parameters.
2) On the gateway, the system looks up the module id provided, and calls getRPCHandler on its module hook.
3) The gateway then reflectively locates the desired function, executes it, and returns the result.
Design Best Practice

In practice, it is usually best to define an interface that is available to both the gateway and designer that defines the functions available for RPC. Then, on both sides, define classes that implement the interface. On the designer side, you will have an "RPC Proxy" class, that simply converts the function calls into calls to moduleInvoke, as described above. On the gateway side, you'll have a fairly normal class that implements the functions, and is returned for getRPCHandler.

This paradigm is demonstrated here, with a simple function that sends someone's name from the designer to the gateway to retrieve a greeting for them:

The RPC Interface - in the common package

```java
public interface ModuleRPC {
    String getGreeting(String firstName);
}
```

The Designer Proxy - in the designer scope

```java
public class ModuleRPCProxy implements ModuleRPC {
    protected Object invoke(String name, Serializable... params) throws Exception {
        return GatewayConnectionManager.getInstance().getGatewayInterface().moduleInvoke(ModuleMeta.MODULE_ID, name, params);
    }
    public String getGreeting(String firstName) {
        return (String) invoke("getGreeting", firstName);
    }
}
```

The Gateway Implementation - in the gateway scope

```java
public class ModuleRPCImpl implements ModuleRPC {
    public String getGreeting(String firstName) {
        return "Hello, " + firstName;
    }
}
```

In the GatewayModuleHook

```java
public Object getRPCHandler(ClientReqSession session, Long projectId) {
    return new ModuleRPCImpl();
}
```

4.7 Gateway to Client Communication (Push Notification)

Somewhat similar to RPC, but in reverse, is the Push Notification System. This system allows the gateway to send messages to clients and designers. The messages are sent through the GatewaySessionManager and can be sent to a specific client or to all currently active sessions.

How Push Notifications Work

Push notification messages are defined three fields: moduleId, messageType, and message. The first two are string fields used to identify the receiver, while the third is a serializable object to be delivered. The gateway places these notifications into a queue for each client session that should receive them, and the client retrieves them at the next gateway poll. The messages are then delivered to any registered listeners in the client.

Receiving Messages in the Designer/Client
To receive push messages in the client, you simply register a PushNotificationListener with the GatewayConnectionManager singleton. For example:

```
GatewayConnectionManager.getInstance().addPushNotificationListener(new MyPushListener);
```

The only problem with this is that the listener will receive all push messages, for all modules, and will need to filter them itself. To help, the FilteredPushNotification base class can be used instead. This class will automatically filter messages based on module id and message type.

**Delivering Messages on the EDT**

It is common that messages received will trigger a change to the UI state of a module. As UI operations should only be performed on the Event Dispatch Thread ("EDT"), the code handling push notifications often contains many calls to `EventQueue.invokeLater()`. The FilteredPushNotification can handle this task for you, however, if you override the `dispatchOnEDT()` method to return true. That way, your implementation of `receive()` does not need to worry about which thread it is on.

**Sending Messages from the Gateway**

**Sessions and the GatewaySessionManager**

Before using the push notification system, it is important to understand the concept of a session. A session, as the name implies, is one particular instance of a client connection, spanning it's lifetime. The term "client" is used to describe both designer connections, and Vision module runtime clients. When a user logs in, a session is created with a unique identifier, and is categorized based on the "scope", or connection type, it represents.

The sessions are managed by the GatewaySessionManager, and can be accessed through that system in the GatewayContext.

**Sending Notifications**

There are two ways to send notifications: to many clients at once through `GatewaySessionManager.sendNotification()`, or to a specific client session, by getting the `ClientReqSession` object from the GatewaySessionManager and calling `addNotification()`. Since the first method is able to filter based on scope, the second method is generally only used in more complex schemes where the client registers itself with the module on the gateway through RPC, in order to provide the gateway with its session id, and the gateway then sends messages specifically for that client.

**Example: Sending a notification to all designers**

Designer sessions are identified with the scope `ApplicationScope.DESIGNER`. Therefore, is easy to send a message to all active designers through the GatewaySessionManager. In this example, we send a message with the id "_IMPORTANT_MSG_" to our module in the designer. The message class, ImportantMessage, is an imaginary serializable message that will be delivered to the other side.

```
context.getGatewaySessionManager().sendNotification(ApplicationScope.DESIGNER, "mymodule_id", "_IMPORTANT_MSG_", new ImportantMessage("Problem detected!", true));
```

**Design Best Practice**

The examples in this chapter use strings directly for the module id and message ids to improve clarity, but as mentioned elsewhere, it is preferable to define these strings as constants in a central "module meta" class.
4.8 Providing Progress Information

Everyone hates it when they perform some action and the system simply locks up for an indeterminate amount of time. This is usually caused by a poorly programmed task executing for longer than expected, on the EDT, which prevents the screen from updating. These situations can be drastically improved by executing the task in its own thread, and providing the user feedback as to the progress of the task. The ClientProgressManager, accessed through the ClientContext, helps with both of these.

The ClientProgressManager allows you to register tasks for execution. When the task executes, it will be in a different thread, and can provide information about its progress. The system will take care of displaying the progress for you. If the task allows cancelling, the user will be given that option. Tasks initiated in the designer/client that pass to the gateway can also support progress, if programmed to do so in the gateway. Progress information generated in the gateway will be passed back to the initiating session for display.

Creating and Executing an AsyncClientTask

To use the progress system, you must first create a task. This is done by implementing the AsyncClientTask interface. This interface defines three functions: one for the task title, one dictating whether the task can be canceled, and a "run" function that actually performs the task. The run function takes a TaskProgressListener which it can update with information, and through which the cancel state will be sent.

Once defined, the task should be registered with the ClientProgressManager.runTask() function. There are two forms of this function, one that takes an additional boolean flag indicating whether the task is dominant. Dominant tasks are given priority for display. Normally, tasks get stacked in the order in which they’re registered. Dominant tasks, however, go to the top. This is useful when creating tasks that spawn other tasks, particularly in the gateway. If your original task is not dominant, when the next task begins it will take the display. If the original task is dominant, however, it will remain in display, and can provide information about the overall progress instead of each individual sub-task.

Updating Progress

The executing task is provided with a TaskProgressListener that it can update with status information. This listener can be used as much as necessary- the note can be updated multiple times, the progress reset, etc. Since very frequently the executing

If the task is cancelable, and has been canceled, it will be reflected through the isCanceled() function. Therefore, the executing task should check this flag frequently and exit gracefully if this situation is encountered.

If the progress of a task cannot be known exactly, the task should be marked as indeterminate. Indeterminate tasks do not return progress, and get displayed appropriately with a bouncing bar or other animation that indicates something is happening.

Tasks are finished when the executing function returns. If the function throws an exception, the error will be displayed to the user. Therefore, it is perfectly normal to throw exceptions when the task cannot be finished for any particular reason that the user should know about.

Tracking Progress on the Gateway

If a client task calls a function on the gateway that supports asynchronous invocation, the gateway can
provide progress information that gets piped back to the calling client. The client can also request that
tasks on the gateway get canceled. Unlike in the designer/client scope where the special interface must
be implemented, in the gateway the progress system is designed to be unobtrusive and compatible with
existing code. To support gateway to client progress, the invoked function must be marked as
asynchronous (using the \texttt{AsyncFunction} annotation), and then the actual executing function must
use the \texttt{GatewayProgressManager.getProgressListenerForThread()} function to get the
listener to update.

For most modules, the only functions that might be used with this system are the RPC functions. For
example, let's imagine that we extend our RPC example to include a function that pings an address from
the gateway multiple times, and returns whether all the attempts were successful. For simplicity's sake,
the ping will be handled by a function that isn't shown:

```java
public class ModuleRPCImpl implements ModuleRPC {

...previous definition...

@AsyncFunction
public boolean testAddress(String addr){
  boolean result=true;
  //Get the progress listener for this thread
  TaskProgressListener progress = gatewayContext.getProgressManager().getProgressListenerForThread();
  progress.setProgressMax(5);
  for(Integer i=0; i<5; i++){
    //Update progress to show current state.
    progress.setProgress(i);
    progress.setNote("Executing test attempt: "+(i+1).toString());

    boolean testResult = ping(addr);
    if(!testResult){
      result=false;
      break;
    }
  }
  //If the task has been canceled, return what we have so far
  if(progress.isCanceled()){break;}

  return result;
}
```
Vision Component Development

Part V
5 Vision Component Development

5.1 Module Structure

One of the easiest types of modules to write is a module that adds a new component to the Vision module. Vision components are written in Java Swing and are modeled after the JavaBeans specification (not to be confused with Enterprise JavaBeans or EJ Bs, which is entirely different). If you've never used Swing before, you should at least go through The Swing Tutorial before getting started.

A basic module that adds components to the Vision module will need two projects: one for the Client scope and one for the Designer scope. You do not need a project for the Gateway scope unless your components are part of a larger module that requires Gateway-scoped resources.

In your Client-scope project you'll have all of your components defined. You don't strictly need a Client-scoped module hook class at all. Your components will get compiled into a jar file that you'll mark in your module.xml file as "DC" - designer and client scope.

In your Designer-scope project you'll have a hook class and your BeanInfo classes. The hook will be responsible for adding your components to the Vision Module's palette. The BeanInfo classes are used to describe the components to the Vision module. Make sure to mark the designer's hook with a dependency on the Vision module, whose module id is "fpmi" because of its historical roots as our legacy FactoryPMI product.

5.2 Component Overview

Writing a component for the Vision module is as easy as writing any Java Swing-based component. There are only a few things specific to Ignition and the Vision module that you need to be aware of.

JavaBean Properties

One of the most fundamental parts of writing a component is exposing properties to the rest of the Vision system. These properties are going to be the primary mechanism for your component to be configured and to interact with data from the rest of Ignition, such as tag data and SQL query data. To expose a property, you simply need to make it accessible via JavaBean-style getter/setter functions.

For example, if I have a String field called title on my component, I'd add these functions to expose the property:

```java
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
```

Now as long as I include the title property in my component's BeanInfo class, it will show up in the Property Table pane when the user selects the component in the designer. That means they can set the value of the property, or bind it to a tag, a query, or whatever. Note that your component remains blissfully unaware of the binding system - all it cares about is that the title field may change, and it knows how to deal with that.

Bound Properties
Components have two major types of properties: normal properties and bound properties. All properties whose type is understood by Ignition can use the binding system. That is, the user can configure a binding on that property. However, only bound properties can be bound to. To understand this, look at the Text Field component. It has many properties. Now put a Label on a window and bind the text of the label to the text field's text. You'll notice that the list of the Text Field's properties that you can bind the label's text to is much shorter than the whole list of properties for the Text Field. This shorter list is the list of bound properties. A better name for these might be: properties that can be bound to.

There are two things that make a property a bound property.

1. In the component's BeanInfo class, the component is marked as a bound property with the `BOUND_MASK`.
2. The component property fires the `propertyChangeEvent` when the field is altered. This event is what powers the binding system. For example, to make the title field fire events, I'd alter the setter function to this:

```java
public void setTitle(String title) {
    String oldValue = this.title;
    this.title = title;
    firePropertyChange("title", oldValue, title);
}
```

### Special Abilities

There are a few special abilities that users of the Vision system expect a component's to have. The ability to add dynamic properties and displaying the correct quality overlay are examples of these. By having your component implement the correct interfaces, they gain these abilities.

To save you the trouble of implementing a handful of interfaces that you aren't familiar with, the Vision module provides abstract base classes that you are encouraged to extend. So, instead of extending directly from `JComponent`, you might extend `AbstractVisionComponent`, which itself extends from `JComponent`. Doing this will get you: dynamic properties, quality overlays, styles, correct cursor and name collision handling. It is highly recommended that you extend from either `AbstractVisionComponent`, `AbstractVisionPanel`, or `AbstractVisionScrollPane` to start your component off on the right foot.

### Lifecycle

All components are expected to implement the `VisionComponent` interface (All of the abstract base classes above already do this for you). This defines the quality monitoring behavior as well as the `ComponentLifecycle` behavior. `ComponentLifecycle` is an interface that defines a startup and shutdown method. The startup method gives you the `VisionClientContext`, which can be handy for referencing the rest of the system. More importantly, it lets your component know when to shut itself down. Any component that has long-running background processes (threads) needs to shut them down when the `shutdownComponent()` function is called. Not doing this will create a resource leak. This function is called when the window that contains the component is closed.

### 5.3 BeanInfo Classes

Each of your components will have a corresponding BeanInfo class. These classes are loaded up in the Designer scope only, and are used to describe the component to the Designer.

### BeanInfo Location
BeanInfo classes are located using a combination of naming conventions and explicit configuration.

BeanInfo classes must always be named the same as the component they describe, with "BeanInfo" on the end. For example, if you've made a component called MyGreatChart, your BeanInfo class for that component would be named MyGreatChartBeanInfo. Typically, you'll have all if your BeanInfo classes residing in a single package in your Designerscoped project. For the sake of example, lets say that package is: com.example.mymodule.beaninfos. Using your module's Designer-scope hook class, you'll add that package name to the Vision module's BeanInfo search path. For example:

```java
public class MyModuleDesignerHook extends AbstractDesignerModuleHook {
    public void startup(DesignerContext context, LicenseState activationState) throws Exception {
        context.addBeanInfoSearchPath("com.example.mymodule.beaninfos");
    }
}
```

Now when the Designer needs to look for the BeanInfo class for the MyGreatChart component, it knows to look for the class:

```
com.example.mymodule.beaninfos.MyGreatChartBeanInfo
```

### Writing a BeanInfo Class

A BeanInfo class describes your component to the Vision module. It is used to present the user with a nice-looking list of properties when your component is selected. The properties get friendly descriptions and are categorized and prioritized. It defines the name and icon for your component's presence in the palette. It can add your own customizers for extended configuration.

Your BeanInfo class itself should extend from CommonBeanInfo. It will need to override methods like `initProperties` in order to all of the properties of your component. Follow the examples in the SDK.

### 5.4 Designer Hook

The designer hook class of a module that adds Vision module properties has two main functions: add the bean info search path and add the components themselves into the Palette.

To get the Palette, you'll need to grab the Vision module's VisionDesignerInterface. You can do this through the DesignerContext. Once you have the VisionDesignerInterface, you can get the palette and make your own PaletteItemGroup. You'll add all of your components to this item group. An example should make this clear:
public class MyModuleDesignerHook extends AbstractDesignerModuleHook {

    public void startup(DesignerContext context, LicenseState activationState) throws Exception {
        context.addBeanInfoSearchPath("com.example.mymodule.beaninfos");

        VisionDesignerInterface sdk = (VisionDesignerInterface) context.getModule(VisionDesignerInterface.VISION_MODULE_ID);
        if (sdk != null) {
            Palette palette = sdk.getPalette();
            PaletteItemGroup group = palette.addGroup("MyModule");
            group.addPaletteItem(new JavaBeanPaletteItem(MyGreatChart.class));
            group.addPaletteItem(new JavaBeanPaletteItem(SomeOtherComponent.class));
        }
    }
}
6  OPC-UA Driver Development

6.1  Three Steps to a UA Driver

6.1.1  Implement the Driver Interface

All OPC-UA drivers must, at a minimum, implement the Driver interface. This interface defines the basic functionality any driver must provide - reading, writing, browsing, subscription management, and life cycle/state management.

A partially complete implementation of Driver is available by extending from AbstractDriver or AbstractNioDriver. Extending from these classes is encouraged as the complexities derived from managing subscription state and the request life cycle of your subscriptions in a highly asynchronous environment are handled for you.

**Driver Best Practice**

Extending from AbstractDriver, or AbstractNioDriver if appropriate, will make your life a whole lot easier.

6.1.2  Implement the DriverMeta Interface

The DriverMeta interface serves two essential functions.

First, it provides the name of your driver class and a ClassLoader with which to load it. Second, it is the source of any configurable properties your driver needs to present to the user, as well as a few pieces of metadata that are displayed in the Ignition Gateway.

**Categories and Properties**

Drivers expose their configurable properties using DriverCategory's and DriverProperty's.

A DriverCategory corresponds to a category on the configuration page that will get generated for your driver when a user adds or edits it. Each DriverCategory contains a list of DriverProperty’s.

DriverProperty’s provide a name, description, default value, and optionally some validation parameters.

Every property a driver needs to have exposed as configurable will need a corresponding DriverProperty implementation. All of these properties must be provided by one or more DriverCategory's.

**Driver Best Practice**

Partial implementations are available in AbstractDriverMeta, AbstractDriverCategory and AbstractDriverProperty. These partial implementations take care of most of the work of externalizing your name and description strings into properties files as well as providing default implementations of some of the less commonly used methods of each interface.
6.1.3 Register Your Driver

Once you have your Driver implementation and DriverMeta implementation you can register your driver with the DriverManager.

Registering your DriverMeta is a three step procedure:

1. Implement the ModuleServiceConsumer interface on your modules hook class. For more information about module hooks, see Scopes, Hooks, and Contexts.

2. During startup(), subscribe to DeviceManager.class.

   ```java
   context.getModuleServicesManager().subscribe(DeviceManager.class, this);
   ```

3. In serviceReady(Class<?> serviceClass), do something like the following:

   ```java
   @Override
   public void serviceReady(Class<?> serviceClass) {
       if (serviceClass == DeviceManager.class) {
           driverManager = (DeviceManager) context.getModuleServicesManager().getService(DeviceManager.class);
           if (driverManager != null) {
               for (DriverMeta meta : driverMetas) {
                   driverManager.registerDriver(meta);
               }
           }
       }
   }
   ```

During shutdown() it’s important to unsubscribe to the DeviceManager.class service as well as unregister any drivers you may have registered.

```java
@Override
public void shutdown() {
    if (context != null) {
        context.getModuleServicesManager().unsubscribe(DeviceManager.class, this);
        if (driverManager != null) {
            for (DriverMeta meta : driverMetas) {
                driverManager.unregisterDriver(meta);
            }
        }
        context = null;
        driverManager = null;
    }
}
```

Driver Best Practice

A partial implementation of a drivers GatewayModuleHook can be found in AbstractDriverModuleHook. This class takes care of all the details of registering and unregistering your driver. Subclasses need only provide a List<DriverMeta> to register.

**Important note:** During the course of startup you should check and make sure DriverAPI.VERSION
is the API version number your driver is written against. Changes in API version number are not meant to be backwards compatible. If you extend from AbstractDriverModuleHook all you need to do is implement getExpectedAPIVersion().

Example:

```java
@Override
public void startup(LicenseState activationState) {
    if (DriverAPI.VERSION != EXPECTED_API_VERSION) {
        throw new RuntimeException(String.format("Expected Driver API version %s but found %s instead.",
                                            EXPECTED_API_VERSION, DriverAPI.VERSION));
    }
    ...
}
```
Examples

Part VII
7 Examples

7.1 Component Example

The Component Example project is a simple project that does one thing: adds a component to the Vision module. It shows everything required to write your own component and add it to a palette, including how to benefit from special Vision features like dynamic properties and quality overlays. Writing your own Vision component(s) requires a basic understanding of Java Swing. Read more about this kind of development in the Vision Component Development section.

Concepts Illustrated by the Example

- **Client and Designer** scoped resources. This example only has two projects, the project that contains the component itself and the project that contains some meta-information about the component for the Designer's benefit.
- **Component Lifecycle**. Components need to be aware if their own lifecycle if they do background tasks, such as animation so that they can shut these tasks down when their window is closed. The component in this example shows how this is done.
- **BeanInfo** classes are used to describe a component to the Designer. These classes define what properties of the component to expose, customizers, event sets, etc so the component can be gracefully configured by a user.
- **Designer Module Hook** class is used to configure the Designer and make the Vision module aware of your component.

How to Use

To use this example, set up your workspace as described under Environment Setup. You'll have two projects in your workspace: `ComponentExample_Client` and `ComponentExample_Designer`. The client project contains a single class - the `HelloWorldComponent`. The designer project contains two classes: the designer module hook and the bean info descriptor for the hello world component. To build this module and deploy it to your running developer-mode Ignition Gateway, run the Ant build file `Build/build-component-example.xml`.

7.2 SQLTag Provider Example

The TagProviderExample project, located in the SDK examples folder, illustrates how a simple SQLTag Provider type can be created to provide data to Ignition through the SQLTags system. There are, unfortunately, some areas where the extensibility of SQLTags is limited, for example the creation of new tag types, though this will be improved in future versions of the SDK. The current goal is to allow modules to provide data as tags, as an alternative to writing the data provider as a driver for the OPC-UA server. It is important to recognize the benefits of writing data providers as a driver, such as the ability to access it from any OPC-UA client and not just Ignition, and the ability to use alerting and SQLTags history when those tags are brought in to a SQLTags provider. Still, in other situations, the ability to (or necessity of) dragging over tags from OPC to SQLTags is undesirable, and you simply want the tags to exist once. In these cases, implement a tag provider is likely the best route.

Concepts Illustrated by the Example

The example demonstrates a few core concepts require to create a provider:

- **Implement TagProvider** - The TagProvider interface defines a realtime SQLTags Provider. The core of the example is a provider which creates tags and gives them random values.
- **Use Extension Points to Register Your Provider** - Tag providers are hooked into the Ignition
gateway through the extension point system, using the gateway's TagManager. The example demonstrates how to declare a tag provider type, and register it.

- **Managing Tag Subscriptions** - The SQLTagsManager holds a central tag subscription model, but it is up to the providers to manage the actual connection between the tag subscriptions and the tags. This example illustrates one of several possible strategies, with a custom tag subscription model that organizes the subscriptions in a way that is convenient for our methodology. It also makes it easy for us to fire changes.

- **Managing Tag Structure** - The example illustrates how tags can be added and removed dynamically, and how the designer can be notified of those changes by firing tag change events for the modified folder.

**Future Improvement**

As mentioned, we recognize that the extensibility of SQLTags can be improved. Shortly in the future, we plan to add:

- Improved configuration support
- The ability to define custom tag types
- The ability to add custom tag properties
- Better base class support so that less needs to be implemented manually
- Zero-work support for alerting and SQLTags history, so that tags published by your provider can be used with these systems automatically.
Endnotes 2... (after index)