Technical Document

Niagara$^{AX-3.5}$ Developer Guide

April 6, 2010

Niagara^{AX} Developer Guide

Confidentiality Notice

The information contained in this document is confidential information of Tridium, Inc., a Delaware corporation ("Tridium"). Such information, and the software described herein, is furnished under a license agreement and may be used only in accordance with that agreement.

The information contained in this document is provided solely for use by Tridium employees, licensees, and system owners; and, except as permitted under the below copyright notice, is not to be released to, or reproduced for, anyone else.

While every effort has been made to assure the accuracy of this document, Tridium is not responsible for damages of any kind, including without limitation consequential damages, arising from the application of the information contained herein. Information and specifications published here are current as of the date of this publication and are subject to change without notice. The latest product specifications can be found by contacting our corporate headquarters, Richmond, Virginia.

Trademark Notice

BACnet and ASHRAE are registered trademarks of American Society of Heating, Refrigerating and Air-Conditioning Engineers. Microsoft and Windows are registered trademarks, and Windows NT, Windows 2000, Windows XP Professional, and Internet Explorer are trademarks of Microsoft Corporation. Java and other Java-based names are trademarks of Sun Microsystems Inc. and refer to Sun's family of Java-branded technologies. Mozilla and Firefox are trademarks of the Mozilla Foundation. Echelon, LON, LonMark, LonTalk, and LonWorks are registered trademarks of Echelon Corporation. Tridium, JACE, Niagara Framework, Niagara^{AX} Framework, and Sedona Framework are registered trademarks, and Workbench, WorkPlace^{AX}, and ^{AX}Supervisor, are trademarks of Tridium Inc. All other product names and services mentioned in this publication that is known to be trademarks, registered trademarks, or service marks are the property of their respective owners.

Copyright and Patent Notice

This document may be copied by parties who are authorized to distribute Tridium products in connection with distribution of those products, subject to the contracts that authorize such distribution. It may not otherwise, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Tridium, Inc.

Copyright © 2010 Tridium, Inc.

All rights reserved. The product(s) described herein may be covered by one or more U.S or foreign patents of Tridium.

Niagara Developer Guide

Framework

- Overview Provides a high level overview of the Niagara Framework and its problem space.
- Architecture Provides a broad overview of the architecture and introduces key concepts such as station, workbench, daemon, fox, and modules.
- API Information Provides an overview of API stability designation and public versus implementation APIs.
- Modules Provides an introduction to modules which are the software deliverables in the Niagara Framework.
- Object Model An introduction to the Niagara type system.
- Component Model An introduction to the Niagara component model.
- Building Simples Details for building simple Types.
- Building Enums Details for building enum Types.
- Building Complexes Details for building complex struct and component Types.
- Registry The Niagara registry is used to query details about what is installed on a given system.
- Naming The Niagara ord naming architecture and its APIs.
- Links Discusses object links.
- Execution Discusses the runtime execution environment.
- Station Discusses station lifecycle.
- Remote Programming Describes how the component model is programmed across the network.
- Files Discusses how files are mapped into the Niagara object model.
- Security Discusses the security model in the Niagara Framework.
- Localization Discusses localization in the Niagara Framework.
- Spy An overview of the Niagara diagnostics framework.
- Licensing An overview of the Niagara licensing framework.
- XML An overview of Niagara XML document model and parser APIs.
- Bog Format Details of the Bog (Baja Object Graph) XML schema and APIs used to encode and decode component graphs.
- Distributions An overview of Niagara Distributions.
- Test How to use Niagara's test framework.
- Virtual Components An overview of using transient, on-demand (virtual) components.

User Interface

- Gx Provides an overview of gx graphic toolkit.
- Bajaui Provides an overview of the widget component toolkit.
- Workbench Overview of the workbench shell.
- Web Overview of web APIs.
- Hx Overview of Hx APIs.
- Px Overview of Px technology.

Hx

- Overview Overview of Hx Architecture
- HxView Details of HxView
- HxOp Details of HxOp
- HxProfile Details of HxProfile
- Events Detail of Events
- Dialogs Details of Dialogs
- Theming Details of Theming

Horizontal Applications

- Control Overview of the control and automation module.
- History Overview to the historical database module.

- Alarm Overview to the alarming module.
- Schedule Overview of the scheduling module.
- Report Overview of the reporting module.

BQL

- BQL Overview of the Baja Query Language.
- BQL Expressions Details on BQL expressions.
- BQL Examples Provides BQL examples.

Drivers

- Driver Framework Overview of the driver framework.
- PointDevicelet For reading and writing proxy points.
- HistoryDevicelet For importing and exporting histories.
- AlarmDevicelet For routing incoming and outgoing alarms.
- ScheduleDevicelet Used to perform master/slave scheduling.
- Basic Driver APIs for the basic driver framework
- BACnet APIs for the BACnet driver
- Lonworks APIs for the Lonworks driver
- Lon Markup Language Specification for the lonml XML format

Development Tools

- Build Documentation on using the build tool to compile and package software modules.
- Deploying Help How to build and package help documentation with Niagara modules.
- Slot-o-matic 2000 Documentation for the slot-o-matic tool used to aid in the generation of boiler plate code for slot definitions.

Architecture Diagrams

- Software Stack Provides an illustration of the major software subsystems in Niagara AX.
- Class Diagram Illustration of the class hierarchy.
- Communication Illustrates Niagara's software processes and their protocols.
- Remote Programming Provides an overview of programming with remote components over fox.
- Driver Hierarchy Illustration of driver hierarchy.
- ProxyExt Illustration of proxy point design.
- Driver Learn Illustration of AbstractManager learn with discovery job.

Niagara Overview

Mile High View

Niagara: a Java software framework and infrastructure with a focus on three major problems:

- Integrating heterogeneous systems, protocols, and fieldbuses;
- Empowering non-programmers to build applications using graphical programming tools;
- Targeted for highly distributed, embedded systems;

Problem Space

Java

The framework uses the Java VM as a common runtime environment across various operating systems and hardware platforms. The core framework scales from small embedded controllers to high end servers. The framework runtime is targeted for J2ME compliant VMs. The user interface toolkit and graphical programming tools are targetted for J2SE 1.4 VMs.

Integrating Heteregenous Systems

Niagara is designed from the ground up to assume that there will never be any one "standard" network protocol, distributed architecture, or fieldbus. Niagara's design center is to integrate cleanly with all networks and protocols. The Niagara Framework standardizes what's inside the box, not what the box talks to.

Programming for Non-programmers

Most features in the Niagara Framework are designed for dual use. These features are designed around a set of Java APIs to be accessed by developers writing Java code. At the same, most features are also designed to be used through high level graphical programming and configuration tools. This vastly increases the scope of users capable of building applications on the Niagara platform.

Embedded Systems

Niagara is targeted for embedded systems capable of running a Java VM. This excludes low devices without 32-bit processors or several megs of RAM. But even embedded systems with the horsepower of low end workstations have special needs. They are always headless and require remote administration. Embedded systems also tend to use solid state storage with limited write cycles and much smaller volume capacities than hard drives.

Distributed Systems

The framework is designed to provide scalability to highly distributed systems composed of 10,000s of nodes running the Niagara Framework software. Systems of this size span a wide range of network topologies and usually communicate over unreliable Internet connections. Niagara is designed to provide an infrastructure for managing systems of this scale.

Component Software

Niagara tackles these challenges by using an architecture centered around the concept of "Component Oriented Development". Components are pieces of self-describing software that can be assembled like building blocks to create new applications. A component centric architecture solves many problems in Niagara:

- Components provide a model used to normalize the data and features of heterogeneous protocols and networks so that they can be integrated seamlessly.
- Applications can be assembled with components using graphical tools. This allows new applications to be built without requiring a Java developer.
- Components provide unsurpassed visibility into applications. Since components are self-describing, it is very easy for tools to introspect how an application is assembled, configured, and what is occurring at any point in time. This provides immense value in debugging and maintaining Niagara applications.
- Components enable software reuse.

Architecture

Overview

This chapter introduces key concepts and terminology used in the Niagara architecture.

Programs

There are typically four different programs (or processes) associated with a Niagara system. These programs and their network communication are illustrated via the Communications Diagram:

- **Station**: is the Niagara runtime - a Java VM which runs a Niagara component application.
- **Workbench**: is the Niagara tool - a Java VM which hosts Niagara plugin components.
- **Daemon**: is a native daemon process. The daemon is used to boot stations and to manage platform configuration such as IP settings.
- **Web Browser**: is standard web browser such as IE or FireFox that hosts one of Niagara's web user interfaces.

Protocols

There are typically three network protocols that are used to integrate the four programs described above:

- **Fox**: is the proprietary TCP/IP protocol used for station-to-station and workbench-to-station communication.
- **HTTP**: is the standard protcol used by web browsers to access web pages from a station.
- **Niagarad**: is the proprietary protocol used for workbench-to-daemon communication.

Platforms

Niagara is hosted on a wide range of platforms from small embedded controllers to high end servers:

- **Jace**: the term Jace (Java Application Control Engine) is used to describe a variety of headless, embedded platforms. Typically a Jace runs on a Flash file system and provides battery backup. Jaces usually host a station and a daemon process, but not workbench. Jaces typically run QNX or embedded Windows XP as their operating system.
- **Supervisor**: the term Supervisor is applied to a station running on a workstation or server class machine. Supervisors are typically stations that provide support services to other stations within a system such as history or alarm concentration. Supervisors by definition run a station, and may potentially run the daemon or workbench.
- **Client**: most often clients running a desktop OS such as Windows or Linux access Niagara using the workbench or a web browser.

Stations

The Niagara architecture is designed around the concept of component oriented programming. Components are self contained units of code written in Java and packaged up for deployment as modules. Components are then wired together to define an application and executed using the station runtime.

A Niagara application designed to be run as a station is stored in an XML file called config.bog. The bog file contains a tree of components, their property configuration, and how they are wired together using links. Station databases can be created using a variety of mechanisms:

- Created on the fly and in the field using workbench graphical programming tools.
- Created offline using workbench graphical programming tools.
- Predefined and installed at manufacturing time.
- Programmatically generated in the field, potentially from a learn operation.

Stations which restrict their programmability to accomplish a dedicated task are often called *appliances*.

Often the term Supevisor or Jace will be used interchangeably with station. Technically the term station describes the component runtime environment common all to all platforms, and Supervisor and Jace describe the hosting platform.

Daemon

The Niagara daemon is the one piece of Niagara written in native code, not Java. The daemon provides functionality used to commission and bootstrap a Niagara platform:

- Manages installing and backing up station databases;
- Manages launching and monitoring stations;
- Manages configuration of TCP/IP settings;
- Manages installation and upgrades of the operating system (QNX only);
- Manages installation and upgrades of the Java virtual machine;
- Manages installation and upgrades of the Niagara software;
- Manages installation of lexicons for localization;
- Manages installation of licenses;

On Windows platforms, the daemon is run in the background as a Window's service. On QNX it is run as a daemon process on startup.

The most common way to access daemon functionality is through the workbench. A connection to the daemon is established via the "Open Platform" command which opens a PlatformSession to the remote machine. A suite of views on the PlatformSession provides tools for accomplishing the tasks listed above.

Another mechanism used to access daemon functionality is via the plat.exe command line utility. This utility provides much of the functionality of the workbench tools, but via a command line program suitable for scripting. Run plat.exe in a console for more information.

Workbench

Niagara includes a powerful tool framework called the workbench. The workbench is built using the bajaui widget framework which is itself built using the standard Niagara component model.

The workbench architecture is designed to provide a common shell used to host plugins written by multiple vendors. The most common type of plugin is a *view* which is a viewer or editor for working with a specific type of object such as a component or file. Other plugins include sidebars and tools.

Workbench itself may be morphed into new applications using the `BWbProfile` API. Profiles allow developers to reuse the workbench infrastructure to create custom applications by adding or removing menu items, toolbar buttons, sidebars, and views.

Web UI

An important feature of Niagara is the ability to provide a user interface via a standard web browser such as IE or FireFox. Niagara provides both server side and client side technologies to build web UIs.

On the server side, the WebService component provides HTTP and HTTPS support in a station runtime. The WebService provides a standard servlet engine. Servlets are deployed as components subclassed from BWebServlet. Additional classes and APIs are built upon this foundation to provide higher level abstractions such as BServletView.

There are two client side technologies provided by Niagara. The first is *web workbench* which allows the standard workbench software to be run inside a web browser using the Java Plugin. The web workbench uses a small applet called wbapplet to download modules as needed to the client machine and to host the workbench shell. These modules are cached locally on the browser's hard drive.

In addition to the web workbench, a suite of technology called hx is available. The hx framework is a set of server side servlets and a client side JavaScript library. Hx allows a real-time user interface to be built without use of the Java Plugin. It requires only web standards: HTML, CSS, and JavaScript.

Fox

The Niagara Framework includes a proprietary protocol called *Fox* which is used for all network communication between stations as well as between Workbench and stations. Fox is a multiplexed peer to peer protocol which sits on top of a TCP connection. The default port for Fox connections is 1911. Fox features include:

- Layered over a single TCP socket connection
- Digest authentication (username/passwords are encrypted)

- Peer to peer
- Request / response
- Asynchronous eventing
- Streaming
- Ability to support multiple applications over a single socket via channel multiplexing
- Text based framing and messaging for easy debugging
- Unified message payload syntax
- High performance
- Java implementation of the protocol stack

API Stack

Niagara provides a broad suite of Java APIs used to customize and extend the station and workbench. The software stack diagram illustrates the various software layers of the architecture:

- **Baja**: The foundation of the architecture is defined via the baja module APIs. These APIs define the basics such as modules, component model, naming, navigation, and security.
- **Horizontal**: Niagara includes an extensive library of prebuilt components applicable to various M2M domains. The modules provide standard components and APIs, including: control, alarming, historical data collection, scheduling, and BQL.
- **Drivers**: Niagara is designed from the ground up to support multiple heterogeneous protocols. Modules designed to model and synchronize data with external devices or systems are called *drivers* and are typically built with the driver framework. Drivers integrate both fieldbus protocols like BACnet and Lonworks as well as enterprise systems like relational databases and web services.
- **Human Interfaces**: An extensive software stack is provided for user interfaces. The gx framework provides a standard model and APIs for low level graphics. Built upon gx is the bajaui module which provides a professional toolkit of standard widgets. Built upon bajaui is the workbench framework which provides the standard APIs for writing plugin tools. The px framework and tools are used to enable non-programmers and developers alike to create new user interfaces via XML.

API Information

Overview

There are a huge number of APIs available which are documented to varying degrees. In working with a specific API there are a couple key points to understand:

- **Stability**: a designation for the maturity of the API and its likelihood for incompatible changes;
- **Baja vs Tridium**: public APIs are published under `java.baja` packages, and implementation specific code is published under `com.tridium`;

Stability

Public APIs are classified into three categories:

- **Stable**: this designation is for mature APIs which have been thoroughly evaluated and locked down. Every attempt is made to keep stable APIs source compatible between releases (a recompile may be necessary). Only critical bug fixes or design flaws are just cause to break compatibility, and even then only between major revisions (such 3.0 to 3.1). This does not mean that stable APIs are frozen, they will continue to be enhanced with new classes and new methods. But no existing classes or methods will be removed.
- **Evaluation**: this designation is for a functionally complete API published for public use. Evaluation APIs are mature enough to use for production development. However, they have not received enough utilization and feedback to justify locking them down. Evaluation APIs will likely undergo minor modification between major revisions (such 3.0 to 3.1). These changes will likely break both binary and source compatibility. However, any changes should be easily incorporated into production code with reasonable refactoring of the source code (such as a method being renamed).
- **Development**: this designation is for code actively under development. It is published for customers who need the latest development build of the framework. Non-compatible changes should be expected, with the potential for large scale redesign.

What is Baja?

Baja is a term coined from **B**uilding **A**utomation **J**ava **A**rchitecture. The core framework built by Tridium is designed to be published as an open standard. This standard is being developed through Sun's Java Community Process as JSR 60. This JSR is still an ongoing effort, but it is important to understand the distinction between Baja and Niagara.

Specification versus Implementation

Fundamentally Baja is an open specification and the Niagara Framework is an implementation of that specification. As a specification, Baja is not a set of software, but rather purely a set of documentation. The Baja specification will include:

- Standards for how Baja software modules are packaged;
- The component model and its APIs;
- Historical database components and APIs;
- Alarming components and APIs;
- Control logic components and APIs;
- Scheduling components and APIs;
- BACnet driver components and APIs;
- Lonworks driver components and APIs;

Over time many more specifications for features will be added to Baja. But what is important to remember is that Baja is only a specification. Niagara is an implementation of that specification. Furthermore you will find a vast number of features in Niagara, that are not included under the Baja umbrella. In this respect Niagara provides a superset of the Baja features.

`javax.baja` versus `com.tridium`

Many features found in Niagara are exposed through a set of Java APIs. In the Java world APIs are grouped together into *packages*, which are scoped using DNS domain names. Software developed through the Java Community Process is usually scoped by packages

starting with `java` or `javax`. The APIs developed for Baja are all grouped under `javax.baja`. These are APIs that will be part of the open Baja specification and maybe implemented by vendors other than Tridium. Using these APIs guarantees a measure of vendor neutrality and backward compatibility.

Software developed by Tridium which is proprietary and outside of the Baja specification is grouped under the `com.tridium` packages. The `com.tridium` packages contain code specific to how Niagara implements the Baja APIs. The `com.tridium` code may or may not be documented. Most often these packages have their components and slots documented (doc=bajaonly), but not their low level fields and methods. In general `com.tridium` APIs should never be used by developers, and no compatibility is guaranteed.

Note: Tridium has developed some APIs under `javax.baja` even though they are not currently part of the Baja specification. These are APIs that Tridium feels may eventually be published through Baja, but are currently in a development stage.

Modules

Overview

The first step in understanding the Niagara architecture is to grasp the concept of modules. Modules are the smallest unit of deployment and versioning in the Niagara architecture. A module is:

- A JAR file compliant with PKZIP compression;
- Contains a XML manifest in meta-inf/module.xml;
- Is independently versioned and deployable;
- States its dependencies on other modules and their versions;

Versions

Versions are specified as a series of whole numbers separated by a period, for example "1.0.3042". Two versions can be compared resulting in equality, less than, or greater than. This comparision is made by comparing the version numbers from left to right. If two versions are equal, except one contains more numbers then it is considered greater than the shorter version. For example:

```
2.0 > 1.0
2.0 > 1.8
2.0.45 > 2.0.43
1.0.24.2 > 1.0.24
```

Every module has two versions. The first is the "bajaVersion" which maps the module to a Baja specification version. If the module is not published under the Baja process then this value is "0". Secondly every module declares a "vendor" name and "vendorVersion". The vendor name is a case insensitive identifier for the company who developed the module and the vendorVersion identifies the vendor's specific version of that module.

Tridium's vendorVersions are formatted as "major.minor.build[.patch]:

- Major and minor declare a feature release such as 3.0.
- The third number specifies a build number. A build number starts at zero for each feature release and increments each time all the softwares modules are built.
- Addition numbers may be specified for code changes made off a branch of a specific build. These are usually patch builds for minor changes and bug fixes.

So the vendorVersion "3.0.22" represents a module of build 22 in Niagara release 3.0. The vendorVersion "3.0.45.2" is the second patch of build 45 in release 3.0.

Manifest

All module JAR files must include a manifest file in "meta-inf/module.xml". The best way to examine the contents of this file is to take an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<module
  name = "control"
  bajaVersion = "1.0"
  vendor = "Tridium"
  vendorVersion = "3.0.20"
  description = "Niagara Control Module"
  preferredSymbol = "c"
>

<dependencies>
  <dependency name="baja" vendor="Tridium" vendorVersion="3.0"/>
```

```
  <dependency name="bajaui" vendor="Tridium" vendorVersion="3.0"/>
  <dependency name="bql" vendor="Tridium" vendorVersion="3.0"/>
  <dependency name="gx" vendor="Tridium" vendorVersion="3.0"/>
  <dependency name="workbench" vendor="Tridium" vendorVersion="3.0"/>
</dependencies>

<dirs>
  <dir name="javax/baja/control" install="runtime"/>
  <dir name="javax/baja/control/enum" install="runtime"/>
  <dir name="javax/baja/control/ext" install="runtime"/>
  <dir name="javax/baja/control/trigger" install="runtime"/>
  <dir name="javax/baja/control/util" install="runtime"/>
  <dir name="com/tridium/control/ui" install="ui"/>
  <dir name="com/tridium/control/ui/trigger" install="ui"/>
  <dir name="doc" install="doc"/>
</dirs>

<defs>
  <def name="control.foo" value="something"/>
</defs>

<types>
  <type name="FooBar" class="javax.baja.control.BFooBar"/>
</types>

</module>
```

Looking at the root `module` element the following attributes are required:

- **name**: The globally unique name of the module. Developers should use a unique prefix for their modules to avoid name collisions. Module names must be one to 25 ASCII characters in length.
- **bajaVersion**: Baja specification version as discussed above.
- **vendor**: The company name of the module's owner.
- **vendorVersion**: Vendor specific version as discussed above.
- **description**: A short summary of the module's purpose.
- **preferredSymbol**: This is used during XML serialization.

All modules must include a `dirs` element, which contains a `dir` subelement for each of the module's content directories. Each `dir` has a **name** attribute which contains a system-home relative file path for a directory in the module, and an **install** attribute which has one of the following values:

- **runtime**: The contents of the directory are to be installed to all hosts.
- **ui**: The contents of the directory are to be installed only to hosts having module content filter levels of "ui" or "doc".
- **doc**: The contents of the directory are to be installed only to hosts having the "doc" module content filter level.

All modules include zero or one `dependencies` element. This element contains zero or more `dependency` elements which enumerate the module's dependencies. Dependencies must be resolved by the framework before the module can be successfully used. Each dependency has one required attribute. The **name** attribute specifies the globally unique name of the dependent module. A dependency may also specify a bajaVersion and/or a vendor version. If the **bajaVersion** attribute is declared then it specifies the lowest bajaVersion of the dependent module required. It is assumed that higher versions of a module are backward compatible, thus any version greater than the one specified in a dependency is considered usable. Likewise the **vendor** and **vendorVersion** may be specified to declare a dependency on a specific implementation of a module. The **vendor** attribute may be specified without the **vendorVersion** attribute, but not vise versa. The required **embed** attribute specifies whether the dependency should be enforced on embedded Niagara devices.

Modules can declare zero or more `def` elements which store String name/value pairs. The defs from all modules are collapsed into a global def database by the registry.

Modules which contain concrete Niagara `BObjects` also include a `types` element. This element includes zero or more `type` elements. Each `type` element defines a mapping between a Baja type name and a Java class name. This definition is specified in the two required attributes **type** and **class**.

Object Model

Niagara Types

The heart of Niagara is its type system layered above Java type system. Niagara Types are monikers to a Java class in a specific module. The interface `javax.baja.sys.Type` is used to represent Types in the Niagara Framework. Every Type is globally identified by its *module name* and its *type name*. As previously discussed, a *module name* globally identifies a Niagara software module. The *type name* is a simple String name which is mapped to a Java class name by the "module.xml" manifest file. Type's are commonly identified using a format of:

```
{module name}:{type name}

Examples:
  baja:AbsTime
  bajaui:TextField
```

Note: to avoid confusion with the various uses of the word *type*, we will use capitalization when talking about a Niagara Type.

BObject

All Java classes which implement a Niagara Type are subclassed from `BObject` . It is useful to compare `Type` and `BObject` to their low level Java counter parts:

Java	Niagara
`java.lang.Object`	`javax.baja.sys.BObject`
`java.lang.Class`	`javax.baja.sys.Type`
`java.lang.reflect.Member`	`javax.baja.sys.Slot` (discussed later)

`Type` and `Slot` capture the concepts of meta-data, while `BObject` provides the base class of Niagara object instances themselves.

BInterface

Java interfaces may be mapped into the Niagara type system by extending `BInterface`. You can query whether a `Type` maps to a class or an interface using the method `isInterface()`.

Classes which implement `BInterface` must also extent `BObject`. All `BInterfaces` class names should be prefixed with "BI".

BObject Semantics

Subclassing from BObject provides some common semantics that all instances of Niagara Types share:

- They all support a `getType()` method.
- Types installed on a system can be extensively queried using the registry.
- All `BObjects` have an icon accessed via `getIcon()`.
- All `BObjects` have a set of agents accessed via `getAgents()`. Most agents are user agents which provide some visualization or configuration mechanism for the `BObject`.

Building BObject

By subclassing `BObject` you make an ordinary Java class into a Nigara Type. You must obey the following rules when creating a Type:

- Types must declare a mapping between their type name and their qualified Java class name in "module.xml". The Java class name must always be prefixed with 'B', but the type name doesn't include this leading 'B'. For example:

  ```
  <type name="FooBar" class="javax.baja.control.BFooBar"/>
  ```

- All Types must override the `getType()` method to return a statically cached `Type` instance created by the `Sys.loadType()`

method:

```
public Type getType() { return TYPE; }
public static final Type TYPE = Sys.loadType(BFooBar.class);
```

Component Model

Introduction

Built upon Niagara's object model is the component model. Components are a special class of `BObjects` used to assemble applications using graphical programming tools.

Slots

Niagara components are defined as a collection of *Slots*. There are three types of slots:

- **javax.baja.sys.Property**: Properties represent a storage location of another Niagara object.
- **javax.baja.sys.Action**: An action is a slot that specifies behavior which may be invoked either through a user command or by an event.
- **javax.baja.sys.Topic**: Topics represent the subject of an event. Topics contain neither a storage location, nor a behavior. Rather a topic serves as a place holder for a event source.

The Java interfaces used to model slots in the Niagara framework are:

```
javax.baja.sys.Slot
   |
   +- javax.baja.sys.Property
   |
   +- javax.baja.sys.Action
   |
   +- javax.baja.sys.Topic
```

Every slot is identified by a String name which is unique within its Type. Slot names must contain ASCII letters or numbers. Other characters may be escaped using "$xx" or "$uxxxx". Refer to `SlotPath` for the formal grammar of slot names and utilities for escaping and unescaping.

Slots also contain a 32-bit mask of flags which provide additional meta-data about the slot. These flag constants are defined in the `javax.baja.sys.Flags` class. Additional meta-data which is not predefined by a flag constant may be specified using `BFacets` which support arbitrary name/value pairs

Slots are either *frozen* or *dynamic*. A frozen slot is defined at compile time within a Type's Java class. Frozen slots are consistent across all instances of a specified Type. Dynamic slots may be added, removed, renamed, and reordered during runtime. The power of the Niagara Framework is in providing a consistent model for both compile time slots and runtime slots. Frozen and dynamic slots are discussed in detail in Building Complexes.

BValue

All values of Property slots are instances of `javax.baja.sys.BValue`. The `BValue` class hierarchy is:

```
javax.baja.sys.BObject
   |
   +- javax.baja.sys.BValue
      |
      +- javax.baja.sys.BSimple
      |
      +- javax.baja.sys.BComplex
        |
        +- javax.baja.sys.BStruct
        |
        +- javax.baja.sys.BComponent
```

`BSimples` are atomic Types in the Niagara Framework, they never contain any slots themselves. The `BComplex` class is used to built

Types which are composed of slots. Every `BComplex` can be recursively broken down into its primitive `BSimples`.

Building BValues

To define new BValues types refer to the following for rules and design patterns:

- Building Simples Details for building `BSimple` Types;
- Building Enums Details for building `BEnum` Types;
- Building Complexes Details for building `BComplex` and `BComponent` Types;

Building Simples

Overview

`BSimple` is the base class for all atomic data types in Niagara. As an atomic data type, `BSimples` store a simple piece of data which cannot be decomposed. All simples are immutable, that is once an instance is created it may never change its state. Concrete subclasses of `BSimples` must meet the following requirements:

- Meet the common rules applicable to all `BObjects`;
- Must declare a `public static final` field named `DEFAULT` which contains a reference to the default instance for the `BSimple`;
- All `BSimples` must be immutable! Under no circumstances should there be any way for an instance of `BSimple` to have its state changed after construction;
- Every concrete subclass of `BSimple` must be declared final;
- Every `BSimple` must implement the `equals()` method to compare the equality of its atomic data;
- Every `BSimple` must implement binary serialization:

```
public abstract void encode(DataOutput out);
public abstract BObject decode(DataInput in);
```

- Every `BSimple` must implement text serialization:

```
public abstract String encodeToString();
public abstract BObject decodeFromString(String s);
```

- Convention is to make constructors private and provide one or more factory methods called `make`.

Example

The following source provides a example:

```
/*
 * Copyright 2000 Tridium, Inc. All Rights Reserved.
 */
package javax.baja.sys;

import java.io.*;

/**
 * The BInteger is the wrapper class for Java primitive
 * int objects.
 */
public final class BInteger
  extends BNumber
{

  public static BInteger make(int value)
  {
    if (value == 0) return DEFAULT;
    return new BInteger(value);
  }

  private BInteger(int value)
  {
    this.value = value;
```

```
  }

  public int getInt()
  {
    return value;
  }

  public float getFloat()
  {
    return (float)value;
  }

  public int hashCode()
  {
    return value;
  }

  public boolean equals(Object obj)
  {
    if (obj instanceof BInteger)
      return ((BInteger)obj).value == value;
    return false;
  }

  public String toString(Context context)
  {
    return String.valueOf(value);
  }

  public void encode(DataOutput out)
    throws IOException
  {
    out.writeInt(value);
  }

  public BObject decode(DataInput in)
    throws IOException
  {
    return new BInteger( in.readInt() );
  }

  public String encodeToString()
    throws IOException
  {
    return String.valueOf(value);
  }

  public BObject decodeFromString(String s)
    throws IOException
  {
    try
    {
      return new BInteger( Integer.parseInt(s) );
```

```
    }
    catch(Exception e)
    {
      throw new IOException("Invalid integer: " + s);
    }
  }

  public static final BInteger DEFAULT = new BInteger(0);

  public Type getType() { return TYPE; }
  public static final Type TYPE = Sys.loadType(BInteger.class);

  private int value;
}
```

Building Enums

Overview

The `BEnum` base class is used to define enumerated types. An enum is composed of a fixed set of int/String pairs called its *range*. The int identifiers are called *ordinals* and the String identifiers are called *tags*. Enum ranges are managed by the `BEnumRange` class.

There are three subclasses of `BEnum`. `BBoolean` is a special case which models a `boolean` primitive. The `BDynamicEnum` class is used to manage weakly typed enums which may store any ordinal and range. Strongly typed enums may be defined at compile time by subclassing `BFrozenEnum`. The Niagara Framework builds a `BFrozenEnum's` range using the following set of introspection rules:

- Meet the common rules applicable to all `BObjects`;
- Meet the common rules applicable to all `BSimples` (although `BFrozenEnum` is not required to declare a `DEFAULT` field);
- Define a set of `public static final` fields which reference instances of the `BFrozenEnum's` range. Each of these `BFrozenEnum` must declare a unique ordinal value. By convention ordinals should start at zero and increment by one. Each of these `BFrozenEnum` must have a type exactly equal to the declaring class.
- There can be no way to create other instances of the `BFrozenEnum` outside of the fields declaring its range. This means no other instances declared in static fields, returned by a static method, or instantable through non-private constructors.
- There must be at least one `BFrozenEnum` declared in the range.
- The default value of a `BFrozenEnum` is the first instance, by convention with an ordinal value of zero.
- By convention a `public static final int` field is defined for each `BFrozenEnum` in the range to provide access to the ordinal value.

Example

The following source provides a complete example of the implementation for `BOrientation`:

```
/*
 * Copyright 2000 Tridium, Inc. All Rights Reserved.
 */
package javax.baja.ui.enum;

import javax.baja.sys.*;

/**
 * BOrientation defines a widget's orientation as
 * either horizontal or vertical.
 */
public final class BOrientation
  extends BFrozenEnum
{

  public static final int HORIZONTAL = 0;
  public static final int VERTICAL = 1;

  public static final BOrientation horizontal = new BOrientation(HORIZONTAL);
  public static final BOrientation vertical = new BOrientation(VERTICAL);

  public Type getType() { return TYPE; }
  public static final Type TYPE = Sys.loadType(BOrientation.class);

  public static BOrientation make(int ordinal)
  {
    return (BOrientation)horizontal.getRange().get(ordinal);
```

```
  }

  public static BOrientation make(String tag)
  {
    return (BOrientation)horizontal.getRange().get(tag);
  }

  private BOrientation(int ordinal) { super(ordinal); }

}
```

Building Complexes

BStructs vs BComponents

BComplex is the base class for both BStruct and BComponent. Classes never subclass BComplex directly (it doesn't support any public or protected constructors). Rather developers subclass from BStruct or BComponent depending on their needs. In general structs are used as complex data types. BStructs can be built only using frozen properties. BComponents support much more flexibility and are built using frozen and dynamic slots of all types:

	BStruct	BComponent
Frozen Property	X	X
Frozen Action		X
Frozen Topic		X
Dynamic Property		X
Dynamic Action		X
Dynamic Topic		X

As you will learn, BComponents are also the basis for many other features such as BOrds, links, and the event model. You may wonder why you would use a BStruct? There are two main reasons. The first is that because of its limited feature set, it is more memory efficient. The other reason is that properties containing BComponents cannot be linked, but BStructs can be (see Links).

Building BComplexes

All concrete subclasses of BComplex must meet the following requirements:

- Meet the common rules applicable to all BObjects;
- Must declare a public constructor which takes no arguments;
- Declare frozen slots using the introspection patterns defined below.

Introspection Patterns

We have discussed how frozen slots are defined at compile time. Let's take a look at the frameworks knows when frozen slots have been declared. Every slot is composed of two or three Java members. A member is the technical term for a Java field, method, or constructor. At runtime the framework uses Java reflection to examine the members of each class, looking for patterns to self-discover slots. These patterns are based on the patterns used by JavaBeans, with significant extensions. Remember introspection is used only to define frozen slots, dynamic slots are not specified in the classfile itself. There is a different pattern for each slot type.

These introspection patterns require a fair amount of boiler plate code. Although it is not too painful to write this code by hand, you may use Slot-o-matic to generate the boiler plate code for you.

Frozen Properties

Rules

Every frozen property must follow these rules:

- Declare a `public static final Property` field where the field name is the property name.
- The property field must be allocated a Property instance using the `BComplex.newProperty()` method. This method takes a set of flags for the property, and a default value.
- Declare a public getter method with JavaBean conventions: `type getCapitalizedName()`.
- Declare a public setter method with JavaBean conventions: `void setCapitalizedName(type v)`.
- The getter must call `BObject.get(Property)`. The method must not perform any addition behavior.
- The setter must call `BObject.set(Property, BObject)`. The method must not perform any additional behavior.
- The only types which may be used in a property are: subclasses of `BValue, boolean, int, long, float, double`, and `String`. The six non-BValue types have special accessors which should be used in the getter and setter implementations.

Semantics

The introspection rules map Property meta-data as follows:

- **Name**: The Property name is the same as the field name.
- **Type**: The Property type is the one declared in the getter and setter methods.
- **Flags**: The Property flags are the ones passed to `newProperty()`.
- **Default Value**: The Property's default value is the instance passed to `newProperty()`.

Example

The following illustrates an example for different property types:

```
// boolean property: fooBar
public static final Property fooBar = newProperty(0, true);
public boolean getFooBar() { return getBoolean(fooBar); }
public void setFooBar(boolean v) { setBoolean(fooBar, v); }

// int property: cool
public static final Property cool = newProperty(0, 100);
public int getCool() { return getInt(cool); }
public void setCool(int v) { setInt(cool, v); }

// double property: analog
public static final Property analog = newProperty(0, 75.0);
public double getAnalog() { return getDouble(analog); }
public void setAnalog(double v) { setDouble(analog, v); }

// float property: description
public static final Property description = newProperty(0, "describe me");
public String getDescription() { return getString(description); }
public void setDescription(String x) { setString(description, v); }

// BObject property: timestamp
public static final Property timestamp = newProperty(0, BAbsTime.DEFAULT);
public BAbsTime getTimestamp() { return (BAbsTime)get(timestamp); }
public void setTimestamp(BAbsTime v) { set(timestamp, v); }
```

Frozen Actions

Rules

Every frozen action must follow these rules:

- Declare a `public static final Action` field where the field name is the action name.
- The action must be allocated an Action instance using the `BComponent.newAction()` method. This method takes a set of flags for the action and an optional default argument.
- Declare a public *invocation* method with the action name. This method must return `void` or a `BObject` type. This method must take zero or one parameters. If it takes a parameter, it should be a `BObject` type.
- Declare a public *implementation* method, which is named do*CapitalizedName*. This method must have the same return type as the invocation method. This method must have the same parameter list as the invocation method.
- The implementation of the invocation method must call `BComponent.invoke()`. No other behavior is permitted in the method.

Semantics

The introspection rules map Action meta-data as follows:

- **Name**: The Action name is the same as the field name.
- **Return Type**: The Action return type is the one declared in the invocation method.
- **Parameter Type**: The Action parameter type is the one declared in the invocation method.
- **Flags**: The Action flags are the ones passed to `newAction()`.

Example

The following illustrates two examples. The first action contains neither a return value nor an argument value. The second declares both a return and argument value:

```
// action: makeMyDay
public static final Action makeMyDay = newAction(0);
public void makeMyDay() { invoke(makeMyDay, null, null); }
public void doMakeMyDay() {  System.out.println("Make my day!"); }

// action: increment
public static final Action increment = newAction(0, new BInteger(1));
public BInteger increment(BInteger v)
  { return (BInteger)invoke(increment, v, null); }
public BInteger doIncrement(BInteger i)
  { return new BInteger(i.getInt()+1); }
```

Frozen Topics

Rules

Every frozen topic must follow these rules:

- Declare a `public static final Topic` field where the field name is the topic name.
- Declare a fire method of the signature: `void fire`*`CapitalizedName`*`(`*`EventType`*`)`.
- The implementation of the fire method is to call `BComponent.fire()`. No other behavior is permitted in the method.

Semantics

The introspection rules map Topic meta-data as follows:

- **Name**: The Topic name is the same as the field name.
- **Event Type**: The Topic event type is the one declared in the fire method.
- **Flags**: The Topic flags are the ones passed to `newTopic()`.

Example

The following code example illustrates declaring a frozen topic:

```
// topic: exploded
public static final Topic exploded = newTopic(0);
public void fireExploded(BString event) { fire(exploded, event, null); }
```

Dynamic Slots

Dynamic slots are not declared as members in the classfile, but rather are managed at runtime using a set of methods on `BComponent`. These methods allow you to add, remove, rename, and reorder dynamic slots. A small sample of these methods follows:

```
Property add(String name, BValue value, int flags);
void remove(Property property);
void rename(Property property, String newName);
void reorder(Property[] properties);
```

Note: You will notice that methods dealing with dynamic slots take a Property, not a Slot. This is because all dynamic slots including

dynamic Actions and Topics are also Properties. Dynamic Actions and Topics are implemented by subclassing `BAction` and `BTopic` respectively.

Registry

Overview

The registry is a term for a small database built by the Niagara runtime whenever it detects that a module has been added, changed, or removed. During the registry build process all the types in all the modules are scanned. Their classfiles are parsed to build an index for the class hierarchy of all the Niagara types available in the installation.

Some of the functions the registry provides:

- query modules installed without opening each jar file
- query class hiearhcy without loading actual Java classes
- query agents registered on a given Type
- map file extensions to their `BIFile` Types
- map ord schemes to their `BOrdScheme` Types
- defs provide a global map of name/value pairs

API

The `Registry` database may be accessed via `Sys.getRegistry()`. Since the primary use of the registry is to interrogate the system about modules and types without loading them into memory, the registry API uses light weight wrappers:

Registry Wrapper	Real McCoy
`ModuleInfo`	`BModule`
`TypeInfo`	`Type`

Agents

An agent is a special BObject type that provides services for other BObject types. Agents are registered on their target types via the module manifest and queried via the Registry interface. Agents are used extensively in the framework for late binding - such as defining views, popup menus, or exporters for specified target types. Typically agent queries are combined with a type filter. For example, to find all the BExporters registered on a given file:

```
AgentFilter filter = AgentFilter.is(BExporter.TYPE);
AgentList exporters = file.getAgents(null).filter(filter);
```

A couple of examples of how an agent type is registered on a target type in the module manifest (module-include.xml):

```
<type name="ValueBinding" class="javax.baja.ui.BValueBinding">
  <agent><on type="bajaui:Widget"/></agent>
  <agent><on type="baja:Value"/></agent>
</type>

<type name="PropertySheet" class="com.tridium.workbench.propsheet.BPropertySheet">
  <agent requiredPermissions="r"><on type="baja:Component"/></agent>
</type>
```

Agents can be registered on a target only for a specific application using the `app` attribute within the `agent` tag. The application name can be queried at runtime via the `AgentInfo.getAppName()` method. Agent application names are used in conjunction with the `getAppName()` method of `BWbProfile` and `BHxProfile`. An example application specific agent:

```
<type name="ApplianceUserManager"  class="appliance.ui.BApplianceUserManager">
  <agent app="demoAppliance">
    <on type="baja:UserService"/>
```

```
    </agent>
  </type>
```

Defs

Module's can declare zero or more defs in their module manifest. Defs are simple String name/value pairs that are collapsed into a single global map by the registry. A good use of defs is to map a device id to a typespec, bog file, or some other metadata file. Then the registry may be used to map devices to Niagara information at learn time.

Since the defs of all modules are collapsed into a single map, it is important to avoid name collisions. Convention is to prefix your defs using module name plus a dot, for example "lonworks."

When using Niagara's standard build tools, defs are defined in your "module-include.xml":

```
<defs>
  <def name="test.a" value="alpha"/>
  <def name="test.b" value="beta"/>
</defs>
```

Use the registry API to query for defs:

```
  String val = Sys.getRegistry().getDef("test.a");
```

Spy

A good way to learn about the registry is to navigate its spy pages.

Naming

Overview

Niagara provides a uniform naming system to identify any resource which may be represented using an instance of `BObject`. These names are called *ords* for **O**bject **R**esolution **D**escriptor. You can think of a ords as URIs on steriods.

An ord is a list of one or more queries separated by the "|" pipe symbol. Each query is an ASCII string formatted as "<scheme>:<body>". The scheme name is a globally unique identifier which instructs Niagara how to find a piece of code to lookup an object from the body string. The body string is opaque and is formatted differently depending on the scheme. The only rule is that it can't contain a pipe symbol.

Queries can be piped together to let each scheme focus on how to lookup a specific type of object. In general absolute ords are of the format: `host | session | space`. Some examples:

```
ip:somehost|fox:|file:/dir/somefile.txt
ip:somehost|fox:1912|station:|slot:/Graphics/Home
local:|module://icons/x16/cut.png
```

In the examples above note that the "ip" scheme is used to identify a host machine using an IP address. The "fox" scheme specifies a session to that machine usually on a specific IP port number. In the first example we identify an instance of a file within somehost's file system. In the second example we identify a specific component in the station database.

The third example illustrates a special case. The scheme "local" which always resolves to `BLocalHost.INSTANCE` is both a host scheme and a session scheme. It represents objects found within the local VM.

APIs

The core naming APIs are defined in the `javax.baja.naming` package. Ords are represented using the `BOrd` class.

Ords may be resolved using the `BOrd.resolve()` or `BOrd.get()` methods. The `resolve` method returns an intermediate `OrdTarget` that provides contextual information about how the ord was resolved. The `get` method is a convenience for `resolve().get()`.

Ords may be absolute or relative. When resolving a relative ord you must pass in a base object. If no base object is specified then `BLocalHost.INSTANCE` is assumed. Some simple examples of resolving an ord:

```
BIFile f1 = (BIFile)BOrd.make("module://icons/x16/cut.png").get();
BIFile f2 = (BIFile)BOrd.make("file:somefile.txt").get(baseDir);
```

Parsing

Ords may be parsed into their constituent queries using the method `BOrd.parse()` which returns `OrdQuery[]`. In many cases you migth cast a OrdQuery into a concrete class. For example:

```
// dump the names in the file path
BOrd ord = BOrd.make("file:/a/b/c.txt");
OrdQuery[] queries = ord.parse();
FilePath path = (FilePath)queries[0];
for(int i=0; i<path.depth(); ++i)
  System.out.println("path[" + i + "] = " + path.nameAt(i));
```

Common Schemes

The following is an informal introduction some common ord schemes used in Niagara.

ip:

The "ip" scheme is used to identify a `BIpHost` instance. Ords starting with "ip" are always absolute and ignore any base which may be

specified. The body of a "ip" query is a DNS hostname or an IP address of the format "dd.dd.dd.dd".

fox:

The "fox" scheme is used to establish a Fox session. Fox is the primary protocol used by Niagara for IP communication. A "fox" query is formatted as "fox:" or "fox:<port>". If port is unspecified then the default 1911 port is assumed.

file:

The "file" scheme is used to identify files on the file system. All file ords resolve to instances of `javax.baja.file.BIFile`. File queries always parse into a `FilePath` File ords come in the following flavors:

- Authority Absolute: "//hostname/dir1/dir2"
- Local Absolute: "/dir1/dir2"
- Sys Absolute: "!lib/system.properties"
- User Absolute: "^config.bog"
- Relative: "myfile.txt"
- Relative with Backup: "../myfile.txt"

Sys absolute paths indicate files rooted under the Niagara installation directory identified via `Sys.getBajaHome()`. User absolute paths are rooted under the user home directory identified via `Sys.getUserHome()`. In the case of station VMs, user home is the directory of the station database.

module:

The "module" scheme is used to access `BIFiles` inside the module jar files. The module scheme uses the "file:" scheme's formating where the authority name is the module name. Module queries can be relative also. If the query is local absolute then it is assumed to be relative to the current module. Module queries always parse into a `FilePath`

```
module://icons/x16/file.png
module://baja/javax/baja/sys/BObject.bajadoc
module:/doc/index.html
```

station:

The "station" scheme is used to resolve the `BComponentSpace` of a station database.

slot:

The "slot" scheme is used to resolve a `BValue` within a `BComplex` by walking down a path of slot names. Slot queries always parse into a `SlotPath`.

h:

The "h" scheme is used to resolve a `BComponent` by its handle. Handles are unique String identifiers for `BComponents` within a `BComponentSpace`. Handles provide a way to persistently identify a component independent of any renames which modify a component's slot path.

service:

The "service" scheme is used to resolve a `BComponent` by its service type. The body of the query should be a type spec.

spy:

The "spy" scheme is used to navigate spy pages. The `javax.baja.spy` APIs provide a framework for making diagnostics information easily available.

bql:

The "bql" scheme is used to encapsulate a BQL query.

Links

Overview

Links are the basic mechanism of execution flow in the Niagara Framework. Links allow components to be wired together graphically by propogating an event on a one slot to another slot. An event occurs:

- When property slot of a `BComponent` is modified.
- When an action slot is invoked.
- When a topic slot is fired.

Links

A link is used to establish an event relationship between two slots. There are two sides to the relationship:

- **Source**: The source of the link is the `BComponent` generating the event either because one its properties is modified or one its topics is fired. The source of a link is always passive in that is has no effect on the component itself.
- **Target**: The target is the active side of the link. The target `BComponent` responds to an event from the source.

A link is established using a property slot on the *target* `BComponent` which is an instance of `BLink` . The `BLink` struct stores:

- **Source Ord**: identifier for the source `BComponent`;
- **Source Slot**: name of the source component's slot;
- **Target Slot**: name of the target component's slot to act upon;

Note: The target ord is not stored explictly in a `BLink` because it is implicitly derived by being a direct child of the target component.

The following table diagrams how slots may be linked together:

Source	Target	Semantics
Property	Property	When source property changes, set the target property
Property	Action	When source property changes, invoke the action
Action	Action	When source action is invoked, invoke target action (action chaining)
Action	Topic	When source action fires, fire target topic
Topic	Action	When source topic fires, invoke the action
Topic	Topic	When source topic fires, fire target topic (topic chaining)

Link Check

Every component has a set of predefined rules which allow links to be established. These rules are embodied in the `LinkCheck` class. Subclasses may override the `BComponent.doLinkCheck()` method to provide additional link checking.

Direct and Indirect Links

Links are constructed as either *direct* or *indirect*. A direct link is constructed with a direct Java reference to its source `BComponent`, source slot, and target slot. A direct link may be created at anytime. Neither the source nor target components are required to be mounted or running. These links must be explicitly removed by the developer. Direct links are never persisted. Examples of creating direct links:

```
target.linkTo("linkA", source, source.slot, target.slot);

  ...or...

BLink link = new BLink(source, source.slot, target.slot);
target.add("linkA", link);
```

```
link.activate();
```

An indirect link is created through indirect names. A BOrd specifies the source component and Strings are used for the source and target slot names. Since an indirect link requires resolution of a BOrd to get its source component, the source is required to be mounted when the link is activated. Indirect links are automatically removed if their source component is unmounted while the link is activated. Examples of creating an indirect link:

```
BLink link = new BLink(BOrd.make("h:77"), "sourceSlot", "targetSlot");
target.add("linkA", link);
```

Note: Links are rarely created programmatically, but rather are configured using the graphical programming tools. The major exception to this rule is building GUIs in code. In this case it is best to establish direct links in your constructor.

Activation

Links exist in either an *activated* or *deactivated* state. When a link is activated it is actively propagating events from the source slot to the target slot. Activated links also maintain a `Knob` on the source component. Knobs are basically a mirror image of a link stored on the source component to indicate the source is actively propagating events over one or more links. When a link is deactivated event propagation ceases and the Knob is removed from the source component.

Activation:

1. Links are activated when the `BLink.activate()` method is called. If the link is indirect, then the source ord must be resolvable otherwise an `UnresolvedException` is thrown.
2. If creating a direct link using the `BComponent.linkTo()` method the link is automatically activated.
3. Enabled links are activated during `BComponent` start. This is how most indirect links are activated (at station boot time).
4. Anytime a `BLink` value is added as a dynamic property on a running `BComponent` it is activated.

Deactivation:

1. Links are deactivated when the `BLink.deactivate()` method is called.
2. Anytime a property with a `BLink` value is removed from `BComponent` it is deactivated and the target property is set back to its default value.
3. Anytime the source component of a active *indirect* link is unmounted, the link is deactivated and removed from the target component.

Execution

Overview

It is important to understand how `BComponents` are executed so that your components play nicely in the Niagara Framework. The Niagara execution model is based upon:

- **Running State**: Every component may be started or stopped.
- **Links**: Links allow events to propagate between components.
- **Timers**: Timers are established using the `Clock` class.
- **Async Actions**: Async actions are an important feature which prevent tight feedback loops.

Running State

Every `BComponent` maintains a *running* state which may be checked via the `BComponent.isRunning()` method. A component may be put into the running state via the `BComponent.start()` method and taken out of the running state via the `BComponent.stop()` method.

By default whenever a `BComponent` is started, all of its descendent components are also started recursively. This behavior may be suppressed using the `Flags.NO_RUN` flag on a property. During startup, any properties encountered with the *noRun* flag set will not be recursed.

Every `BComponent` may add its component specific startup and shutdown behavior by overriding the `started()` and `stopped()` methods. These methods should be kept short; any lengthy tasks should be spawned off on another thread.

Note: Developers will rarely call `start()` and `stop()` themselves. Rather these methods are automatically called during station bootup and shutdown. See Station Bootstrap.

Links

The primary mechanism for execution flow is via the link mechanism. Links provide a powerful tool for configuring execution flow at deployment time using Niagara's graphical programming tools. Developers should design their components so that hooks are exposed via property, action, and topic slots.

One of the requirements for link propagation is normalized types. Therefore Niagara establishes some standard types which should be used to provide normalized data. Any *control point* data should use one of the standard types found in the `javax.baja.status` package.

Timers

Niagara provides a standard timer framework which should be used by components to setup periodic and one-shot timers. Timers are created using the `schedule()` and `schedulePeriodically()` methods on `Clock`. Timer callbacks are an action slot. The `BComponent` must be mounted and running in order to create a timer.

There are four types of timers created with four different methods on Clock. Two are one-shot timers and two are periodic timers. The difference between the two one-shots and periodic timers is based on how the timers drift. Refer to the Clock bajadoc for more information.

Async Actions

The Niagara execution model is *event based*. What this means is that events are chained through link propagation. This model allows the possibility of *feedback loops* when a event will loop forever in a cyclical link chain. To prevent feedback loops, component which might be configured with cyclical links should use *async actions*. An async action is an action slot with the `Flags.ASYNC` flag set.

Normal actions are invoked immediately either through a direct invocation or a link propagation. This invocation occurs on the callers thread synchronously. On the other hand, async actions are designed to run asynchronously on another thread and immediately return control to the callers thread. Typically async actions will coalesce multiple pending invocations.

By default async actions are scheduled by the built in engine manager. The engine manager automatically coalesces action invocations, and schedules them to be run in the near future (100s of ms). Thus between actual execution times if the action is

invoked one or one hundred times, it is only executed once every execution cycle. This makes it a very efficient way to handle *event blasts* such as dozens of property changes at one time. However all timer callbacks and async actions in the VM share the same engine manager thread, so developers should be cautious not to consume this thread except for short periods.

Niagara also provides a hook so that async actions may be scheduled by subclasses by overriding the `post()` method. Using this method subclasses may schedule the action using their own queues and threads. A standard library for managing invocations, queues, and threads is provided by the following utility classes:

- `Invocation`
  ;
- `Queue`
  ;
- `CoalesceQueue`
  ;
- `Worker`
  ;
- `ThreadPoolWorker`
  ;
- `BWorker`
  ;
- `BThreadPoolWorker`
  ;

System Clock Changes

Some control algorithms are based on absolute time, for example a routine that runs every minute at the top of the mintue. These algorithms should ensure that they operate correctly even after system clock changes using the callback `BComponent.clockChanged(BRelTime shift).`

Station

Overview

A station is the main unit of server processing in the Niagara architecture:

- A station database is defined by a single .bog file `"file:!stations/{name}/config.bog"`;
- Stations are booted from their `config.bog` file into a single VM/process on the host machine;
- There is usually a one to one correspondance between stations and host machines (Supervisors or Jaces). However it is possible to run two stations on the same machine if they are configured to use different IP ports;

Bootstrap

The following defines the station boot process:

1. **Load**: The first phase of bootstrap is to deserialize the `config.bog` database into memory as a `BStation` and mount it into the ord namespace as `"local:|station:"`.
2. **Service Registration**: Once the bog file has been loaded into memory and mounted, the framework registers all services. Services are defined by implementing the `BIService`. After this step is complete each service from the bog file may be resolved using the `Sys.getService()` and `Sys.getServices()` methods.
3. **Service Initialization**: Once all services are registered by the framework, each service is initialized via the `Service.serviceStarted()` callback. This gives services a chance to initialize themselves after other services have been registered, but before general components get started.
4. **Component Start**: After service initialization the entire component tree under `"local:|station:"` is started using `BComponent.start()`. This call in turn results in the `started()` and `descendentsStarted()` callbacks. Once this phase is complete the entire station database is in the running state and all active links continue propagation until the station is shutdown.
5. **Station Started**: After all the components under the BStation have been started, each component receives the `stationStarted()` callback. As a general rule, external communications should wait until this stage so that all components get a chance to initialize themselves.
6. **Steady State**: Some control algorithms take a few seconds before the station should start sending control commands to external devices. To handle this case there is a built-in timer during station bootstrap that waits a few seconds, then invokes the `BComponent.atSteadyState()` callback. The steady state timer may be configured using the "nre.steadystate" system property. Use `Sys.atSteadyState()` to check if a station VM has completed its steady state wait period.

Remote Programming

Overview

Remote programming is one of the most powerful features of Niagara. It is also the number one cause of confusion and performance problems. The term *remote programming* broadly applies to using the component model across a network connection. Some topics like subscription are critical concepts for many subsystems. But most often remote programming applies to programming with components in the workbench across a fox connection to a station (illustration).

The component model provides a number features for network programming:

- Lazy loading of a component tree across the network;
- Automatic synchronization of database tree structure over network;
- Ability to subscribe to real-time property changes and topic events;
- Ability to invoke an action over the network like an RPC;
- Support for timed subscriptions called leasing;
- Automatic support for propagating components changes over network;
- Ability to batch most network calls;

Fundamentals

The component model has the ability to make remote programming virtually transparent. In this diagram, the component "/a/b" is accessed in the workbench VM, but actually lives and is executing in the station VM. The instance of the component in the workbench is called the *proxy* and the instance in the station is called the *master*.

The first thing to note in Niagara is that both the proxy and master are instances of the same class. This is unlike technologies such as RMI where the proxy is accessed using a special interface. Also unlike RMI and its brethren, nothing special is required to make a component remote accessible. All Niagara components are automatically remotable by virtue of subclassing BComponent.

From an API perspective there is no difference between programming against a proxy or a master component. Both are instances of the same class with the same methods. However, sometimes it is important to make a distinction. The most common way to achieve this is via the `BComponent.isRunning()` method. A master component will return true and a proxy false. Although `isRunning()` is usually suitable for most circumstances, technically it covers other semantics such as working offline. The specific call for checking proxy status is via `BComponent.getComponentSpace().isProxyComponentSpace()`.

Note that proxy components receive all the standard change callbacks like `changed()` or `added()`. Typically developers should short circuit these callbacks if the component is not running since executing callback code within a proxy can produce unintended side effects.

Proxy Features

The framework provides a host of features which lets you program against a proxy component transparently:

- The proxy can maintain the state of the master by synchronizing all properties in real-time;
- Actions on the proxy act like RPCs;
- Any changes to the proxy are automatically propagated to the master;

The framework provides the ability to keep a proxy's properties completely synchronized in real-time to the master using *subscription*. While subscribed all property changes are immediately reflected in the proxy. This enables easy development of user interfaces that reflect the current state of a component. Note that only properties support this feature - other fields of your class will not be synchronized, and likely will be invalid if they are populated via station execution. Subscription is covered in more detail later.

Another feature of Niagara is that all actions automatically act like RPCs (Remote Procedure Calls). When you invoke an action on a proxy, it automatically marshals the argument across the network, invokes the action on the master, and then marshals the result back to the proxy VM. Note that all other methods are invoked locally.

Perhaps the most powerful feature of proxies is the ability to transparently and automatically propagate proxy side changes to the master. For example when you set a property on a proxy, it actually marshals the change over the network and makes the set on the master (which in turn synchronizes to the proxy once complete). This functionality works for all component changes: sets, adds, removes, renames, reorders, flag sets, and facet sets. Note that if making many changes it is more economical to batch the changes using a Transaction; this is discussed later.

Proxy States

A proxy component exists in three distinct states:

- **Unloaded**: in this state the proxy has not even been loaded across the network.
- **Loaded**: in this state the proxy is loaded across the network and is known to the proxy VM; it may or may not be out-of-date with the master.
- **Subscribed**: in this state the proxy is actively synchronized with the master.

When a session is first opened to a station, none of the components in the station are known in the workbench. Rather components are lazily loaded into the workbench only when needed. Components which haven't been loaded yet are called *unloaded*.

Components become *loaded* via the `BComplex.loadSlots()` method. Components must always be loaded according to their tree structure, thus once loaded it is guaranteed that all a component's ancestors are also loaded. Rarely does a developer use the `loadSlots()` method. Rather components are loaded as the user expands the navigation tree or a component is resolved by ord.

A loaded component means that a proxy instance representing the master component has been created in the workbench. The proxy instance is of the same class as the master, and occupies a slot in the tree structure identical to the master (remember all ancestors must also be loaded). The proxy has the same *identity* as the master. That means calling methods such as `getName()`, `getHandle()`, and `getSlotPath()` return the same result. However, note that the absolute ords of a proxy and master will be different since the proxy's ord includes how it was accessed over the network (see diagram).

Once a proxy component has been loaded, it remains cached in the loaded state until the session is closed. Loaded proxies maintain their structure and identity automatically through the use of NavEvents. NavEvents are always routed across the network to maintain the proxy tree structure independent of the more fine grained component eventing. For example if a loaded component is renamed, it always reflects the new name independent of subscription state. Or if removed it is automatically removed from the cache.

Loaded components provide a cache of structure and identity, but they do not guarantee access to the current state of the master via its properties. The *subscribed* state is used to synchronize a proxy with it's master. Subscription is achieved using a variety of mechanisms discussed next. Once subscribed a component is guaranteed to have all its property values synchronized and kept up-to-date with the master. Subscription is an expensive state compared to just being loaded, therefore it is imporant to unsubscribe when finished working with a proxy.

Subscription

Subscription is a concept used throughout the framework. Components commonly model entities external to the VM. For example, proxy components model a master component in the station VM. Likewise, components in a station often model an external system

or device. Keeping components synchronized with their external representations is usually computationally expensive. Therefore all components are built with a mechanism to be notified when they really need to be synchronized. This mechanism is called *subscription*.

Subscription is a boolean state. A component can check it's current state via the `BComponent.isSubscribed()` method. The `subscribed()` callback is invoked when entering the subscribed state, and `unsubscribed()` when exiting the subscribed state. The subscribed state means that something is currently interested in the component. Subscribed usually means the component should attempt to keep itself synchronized through polling or eventing. The unsubscribed state may be used to disable synchronization to save CPU, memory, or bandwidth resources.

Subscriptions often chain across multiple tiers. For example when you subscribe to a component in the workbench, that subscribes to the master in a station. Suppose the station component is a proxy point for a piece of data running in a Jace. That causes a subscription over the station-to-station connection resulting in the Jace's component to be subscribed. If the Jace component models an external device, that might initiate a polling operation. Keep in mind that n-tier subscribes might introduce delays. The stale status bit is often used with subscription to indicate that a value hasn't yet been updated from an external device.

A component is moved into the subscribed state if any of the following are true:

- If the component is running and any slot in the component is used as the source of an active link: `isRunning() && getKnobs().length > 0`.
- There are one or more active `Subscribers`.
- The component is permanently subscribed via the `setPermanentlySubscribed()` method. A typical example is a control point with an extension that returns true for `requiresPointSubscription()`.

Collectively these three cases are used by the framework to indicate interest in a component. The framework does not make a distinction between how a component is subscribed, rather all three cases boil down to a simple boolean condition: subscribed or unsubscribed.

The `Subscriber` API is the standard mechanism to register for component events. You can think of Subscriber as the BComponent listener API. Subscriber maintains a list of all the components it is subscribed to, which makes cleanup easy via the `unsubscribeAll()` method. Subscribers receive the `event()` callback for any component event in their subscription list. Note that workbench developers typically use `BWbComponentView` which wraps the Subscriber API and provides automatic cleanup.

Leasing

A common need is to ensure that a component is synchronized, but only as a snapshot for immediate use. The framework provides a feature called *leasing* to handle this problem. A lease is a temporary subscription, typically for one minute. After one minute, the component automatically falls back to the unsubscribed state. However, if the component is leased again before the minute expires, then the lease time is reset.

Leasing is accomplished via the `BComponent.lease()` method.

Batch Calls

Although the framework provides a nice abstraction for remote programming, you must be cognizant that network calls are occuring under the covers and that network calls are extremely expensive operations. The number one cause of performance problems is too many round robin network calls. The golden rule for remote programming is that one large batch network call is almost always better performing than multiple small network calls. Niagara provides APIs to batch many common operations.

Batch Resolve

The first opportunity to batch network calls is when resolving more than one ord to a component. Resolving a component deep down in the tree for the first time requires loading the component and all it's ancestors across the network. And if the ord is a handle ord, a network call is needed to translate the handle into a slot path. The most efficient way to batch resolve is the via the `BatchResolve` API.

Batch Subscribe

Subscription is another key area to perform batch network calls. There are three mechanisms for batch subscribe:

1. The first mechanism is to subscribe using a depth. The common case for subscription is when working with a subsection of

the component tree. Depth based subscribe allows a component and a number of descendent levels to be subscribed via one operation. For example if working with the children and grandchildren of a component, then subscribe with a depth of 2.

2. On rare occasions you may need to subscribe to a set of components scattered across the database. For this case there is a subscribe method that accepts an array of BComponents. Both the Subscriber and BWbComponentView classes provide methods that accept a depth or an array.
3. The third mechanism for batch subscribe is do a batch lease. Batch leasing is accomplished via the static `BComponent.lease()` method.

Transactions

By default, when making changes to a proxy component, each change is immediately marshaled over the network to the master. However, if making many changes, then it is more efficient to batch these changes using `Transaction`. Note most Transactions are used to batch a network call, but do not provide atomic commit capability like a RDBMS transaction.

Transactions are passed as the Context to the various change methods like `set()` or `add()`. Instead of committing the change, the change is buffered up in the Transaction. Note that Transaction implements `Context` and is a `SyncBuffer`. Refer to `Transaction's` class header documentation for code examples.

Debugging

The following provides some tips for debugging remote components:

The spy pages provide a wealth of information about both proxy and master components including their subscribe state. A component spy's page also contains information about why a component is subscribed including the knobs and registered Subscribers. Note that right clicking a proxy component in the workbench causes a local lease, so it does introduce a Heisenberg effect; one work around is to bookmark the spy page to avoid right clicks.

The outstanding leases of a VM can be accessed via the LeaseManager spy page.

The most common performance problem is not batching up network calls. The mechanism for diagnosis is to turn on fox tracing. Specially the "fox.broker" log will illustrate network calls for loads, subscribes (sub), unsubscribes (unsub), and proxy side changes (syncToMaster). The simplest way to turn on this tracing is Log Setup spy page.

Files

Overview

The Niagara Framework is built upon the fundamental principle that everything of interest is modeled as a `BObject`. Files are one of the most basic entities which are mapped into the object model.

The Niagara file model is a comprehensive architecture for mapping all files into a consistent set of APIs:

- Files on the local file system (`java.io.File`);
- Files stored within modules and zip files;
- Files over the network using Fox;
- Files over the network using HTTP;
- Files over the network using FTP;
- Files in memory;
- Files which are autogenerated;

API

The `javax.baja.file` package provides the core APIs used for file acess. There are three core concepts in the file model:

1. `BIFile`: represents a file. In general file extensions are mapped to specific Types of `BIFile` using the registry. Effectively the Niagara Type wraps the MIME type. For example common file types include `file:TextFile`, `file:XmlFile`, `file:ImageFile`, `file:WordFile`. The "file" module contains mappings for common file extensions.
2. `BIFileStore`: models a `BIFile` backing store. For example a `file:TextFile` might exist on the file system, in a zip file, or over a network. Each of these file storage mechanism reads and writes the file differently. There a `BIFileStore` for every `BIFile` which may be accessed via the `BIFile.getStore()` method. Common store types include `baja:LocalFileStore`, `baja:MemoryFileStore`, and `baja:ZipFileEntry`.
3. `BFileSpace`: represents a set of files with a common storage model. `BFileSpaces` are responsible for resolving `FilePaths` into `BIFiles`. The prototypical file space is the singleton for local file system `BFileSystem`. The ord "local:|file:" always maps to `BFileSystem.INSTANCE`.

Mapping File Extensions

You can create custom file types for specific file extensions by following these rules:

- Create an implementation of `BIFile`. Utilize one of the existing base classes such as `baja:DataFile`. If you wish to utilize agents such as file text editors then you must extent `file:TextFile` or at least implement `file:ITextFile`.
- Make sure you override `getMimeType()` to return the MIME type for the file's contents:

```
public String getMimeType() { return "text/html"; }
```

- Provide a custom icon if you wish by overriding the `getIcon()` method:

```
public BIcon getIcon() { return icon; }
private static final BIcon icon = BIcon.std("files/html.png");
```

- Map one of more file extensions to your type using in "module-include.xml":

```
<type name="HtmlFile" class="javax.baja.file.types.text.BHtmlFile">
  <file>
    <ext name="html"/>
    <ext name="htm"/>
  </file>
</type>
```

Security

Overview

Security in the Niagara framework covers a couple of broad topics:

- **Authentication**: Logging in and verifying a user;
- **Encryption**: When and how to use cryptography to secure data;
- **Categories**: Categorizing objects we wish to protect via the security model;
- **Permissions**: Configuring and verifying user permissions on objects through categories;
- **Auditing**: Logging user actions to create an audit trail;

The following steps are used to setup a Niagara security model:

1. First we have to define the users, which are modeled as BUsers.
2. We have to authenticate users, to make sure they are who they say they are. This is done via a login with a username and password.
3. We have to determine what each user can do with each object. The objects we typically wish to protect are Components, Files, and Histories. Each of these objects is categorized into one or more categories.
4. We grant each user a set of permissions in each category. This defines exactly what each user can do with each object in the system.
5. Last we audit anything a user does for later analysis.

Users

The BUser component models security principles in a Niagara system. Typically `BUsers` map to a human user, but can also be used to represent machine accounts for machine to machine logins.

The BUserService is used to store and lookup BUsers during login. The default implementation of `BUserService` simply stores the system users as dynamic slots. You can alternatively use BLdapUserService to lookup users via the LDAP protocol.

`BUser` is used to store the authentication credentials, permissions, as well as any other required meta-data for each user. As a developer if you wish to add additional meta-data to users, then you might consider declaring your own BIMixIn.

Authentication and Encryption

All authentication in the Niagara framework is based on the BUserService configured for a station database. The `BUserService` is used to lookup BUsers by username. The `BUser` is then used to check passwords and security permissions. Encryption is usually negotiated at authentication time.

There are three primary authentication points in the Niagara system:

1. **Fox Workbench to Station**: When a connection is made from workbench to a station, the user is prompted for a username and password which is used to authenticate the Fox connection.
2. **Fox Station to Station**: When a connection is made from a station to another station, a preconfigured username/password is used to authenticate the Fox connection. These credentials are stored in the `NiagaraStation.clientConnection` component.
3. **HTTP Browser to Station**: When a browser hits a station URL, an HTTP authentication mechanism is used to validate the user.

Fox Authentication

The Fox authentication mechanism is configurable via the `FoxService` (usually under the `NiagaraNetwork`). Authentication can be either basic or digest. Digest is the preferred mechanism since the password is never passed in cleartext. However if using LDAP, then basic authentication must be used.

HTTP Authentication

The BWebService is used to manage HTTP requests to a Niagara station. The HTTP realm is always the value of

BStation.stationName. The default authentication mechanism used by the WebService is cookie based authentication. Cookie authentication is required if using the workbench with the Java plugin. If you are deploying an application without web workbench support, you can change the authentication mechaism via the WebService.authenticationScheme property.

You can use the guest account to provide access to some or all of your station without requiring a web login. If the guest account is enabled, then any object the guest has operator read access to may be accessed via the web UI without a logon. As soon as an object is accessed which guest does not have operator read on, then the logon challenge is issued. Set the guest user to disabled to turn this feature off.

Categories

All objects designed to be protected by the security model implement the BIProtected interface. The BIProtected interface extends from the BICategorizable interface. An ICategorizable object has the ability to be assigned to one or more categories. In essense a category is just a number: Category 1, Category 2, Category 3, etc. You can give meaningful names categories by mapping category numbers to a BCategory component within the BCategoryService. Most objects of interest implement the BIProtected interface including BComponent, BIFile, and BIHistory.

Categories are just arbitrary groups - you can use categories to model whatever your imagination dreams up. Typically for security they will map to some type of role, for example any device associated with lighting may be assigned to a "lighting" category. But that same device may also be assigned to a "floor3" category.

Categories are implemented as variable length bit strings with each bit representing a category number: bit 1 for Category 1, bit 2 for Category 2, etc. This bit mask is encapsulated via the BCategoryMask class. CategoryMasks are stored and displayed as hex strings, for example the mask for membership in category 2 and 4 would be "a". There are two special CategoryMasks, the "" empty string represents the NULL mask (membership in no categories) and "*" represents the WILDCARD mask (membership in all categories).

The BICategorizable interface provides a getCategoryMask() method to get the *configured category mask* for the object. However most objects support the notation of category inheritence, where the configured mask is null and the applicable category mask is inherited from an ancestor. This is called the *applied category mask* and is accessed via the getAppliedCategoryMask() method.

Permissions

Once a user has been authenticated, the user is granted or denied permissions for each protected object in the system using the user's configured BPermissionsMap. This map grants the user permissions for each category, thereby granting the user permissions for objects assigned to that category. Users may be configured as *super users* by setting their permissions map to BPermissionsMap.SUPER_USER. Super users are automatically granted every permission in every category for every object.

Permission Levels

Niagara defines two *permission levels* called **operator** and **admin**. Each slot in a BComponent is assigned to be operator or admin based on whether the Flags.OPERATOR bit is set.

Permissions

Each slot is defined as admin or operator level. Six *permissions* are derived to control access to slots:

- **Operator-Read**: Allows the user to view operator level information;
- **Operator-Write**: Allows the user to change operator level information;
- **Operator-Invoke**: Allows the user to view and invoke operator level operations;
- **Admin-Read**: Allows the user to view admin level information;
- **Admin-Write**: Allows the user to change admin level information;
- **Admin-Invoke**: Allows the user to view and invoke admin level operations;

The BPermissions class is used to store a bitmask of these six permissions.

Component Permission Semantics

The following are the standard semantics applied to BComponents:

Operation	On Slot	Permission Required
read	operator non-BComponent properties	operatorRead

write	operator non-BComponent properties	operatorWrite
read	admin non-BComponent properties	adminRead
write	admin non-BComponent properties	adminWrite
read	operator BComponent properties	operatorRead on child
read	admin BComponent properties	operatorRead on child
invoke	operator actions	operatorInvoke
invoke	admin actions	adminInvoke
read	operator topics	operatorRead
read	admin topics	adminRead

Note that the permissions required to access a property containing a BComponent are based on the child BComponent regardless of access to its parent or whether the containing slot is marked operator or admin.

File Permission Semantics

`BIFiles` use the operatorRead permissions to check read access for the file and operatorWrite to check write access. For a directory operatorRead is required to list the directory, and operatorWrite to create a new file.

Computing Permissions

To check the permissions available for a specific object use the `BIProtected.getPermissions(Context)` method. If working with an OrdTarget, then it is preferable to use `OrdTarget.getPermissionsForTarget()`, which computes the permissions once and then caches the result.

The standard mechanism to compute permissions by an `IProtected` object is:

1. If the Context is null or doesn't specify a user, then return `BPermissions.all`
2. Route to `BUser.getPermissionsFor()`. Note: don't use this method directly, because it might by-pass special cases within `IProtected.getPermissionsFor()` (see below).
3. Get the object's mask using `getAppliedCategoryMask()`.
4. Map the category mask to a permissions mask via `BPermissionsMap.getPermissions(BCategoryMask)`, which is a logical "OR" of each permission asssigned to the configured categories.

There are a couple special cases to note. First is that `BComponent` access requires access to the entire ancestor tree. For example to access "c" in "/a/b/c", requires at least operatorRead access to "a" and "b". The system will automatically grant operatorRead to all ancestors of a component which a user has at least one permission on. Note that this calculation is only done periodically, but can be forced using the `CategoryService.update` action.

Another special case is `BIFile` which applies these special rules for file system protection:

1. Files in a BModule are automatically granted operatorRead (this does not include .class files which are never mapped into the ord name space).
2. If the user is not a super user, automatically deny any permissions outside of the station home directory
3. Any remaining cases map to user's configured permissions via the file's categories

Checking Permissions

Permission checks are built-in at several layers of the framework:

- Checked on the BComponent modification methods.
- Checked on all Fox network traffic.
- Access in Workbench.

Each of these checks is discussed in detail.

BComponent Modification

The following methods will check user permissions if a non-null Context is passed with a non-null BUser. If the permission is not available then a PermissionException is thrown.

- **set():** If the property is operator, then must have operator write, otherwise admin write of the containing BComponent.
- **setFlags():** Must have admin write of containing BComponent.
- **add():** Must have admin write.
- **remove():** Must have admin write.
- **rename():** Must have admin write.
- **reorder():** Must have admin write.
- **invoke():** If the action is operator, then must have operator invoke, otherwise admin invoke.

Developers should take care to use the proper version of the method with a user context when applicable.

Fox Traffic

Fox is the primary protocol used for workbench-to-station and station-to-station communication. Fox automatically performs all permission checks on the server side before sensitive data can be accessed or modified by a client. By the time a BComponent reaches the client Fox ensures the following:

- Dynamic slots which the user lacks permission to read are never sent across the network and will never appear in the client.
- Frozen slots which is the user lacks permission to read/invoke will automatically have the hidden flag set.
- Frozen properties which the user lacks permission to write will automatically have the readonly flag set.

Furthermore all attempts to modify components are checked by the server being committed.

Workbench Access

Each view declares the permissions a user is required to have on a given BComponent in order to access the view. These permissions are usually declared in the module manifest (module-include). By default views require adminWrite. To override the default:

```
<type name="PropertySheet" class="com.tridium.workbench.propsheet.BPropertySheet">
  <agent requiredPermissions="r"><on type="baja:Component"/></agent></type>
```

Note that required permissions for a dynamic `PxViews` are configured via the `BPxView.requiredPermissions` property.

Auditing

One of the important aspects of security is the ability to analyze what has happened after the fact. The Niagara component model is designed to audit all property modifications and action invocations. Auditable actions include:

- Property changed
- Property added
- Property removed
- Property renamed
- Properties reordered
- Action invoked

Component modifications are only audited when the modification method is passed a non-null Context with a non-null BUser. The history module includes a standard implementation of an audit trail stored to a history database file.

Code Samples

In order to check if a BUser has a operator read permission on specified component:

```
target.getPermissionsFor(user).has(BPermissions.operatorRead) // BUser implements Context
```

This snippet of code will throw a PermissionException if the user lacks the admin invoke permission:

```
user.check(target, BPermissions.adminInvoke)
```

To filter a list of `INavNode` children for security:

```
BINavNode[] kids = node.getNavChildren();
kids = BNavContainer.filter(kids, context);
```

Use an AccessCursor to automatically skip slots that a user lacks permission to read/invoke:

```
SlotCursor c = AccessSlotCursor.make(target.getSlots(), user)
while(c.next()) {}
```

Localization

Overview

All aspects of the Niagara framework are designed for localization. The basic philosophy for localization is that one language may supported in-place or multiple languages may be supported via indirection. The foundation of localization is based on the Context and Lexicon APIs.

Context

Any framework API which is designed to return a string for human display, takes a Context parameter. Context provides information to an API about the context of the call including the desired locale. Many APIs implement Context directly including OrdTarget, ExportOp, and WebOp. For example if you are processing a web HTTP request, you can pass the WebOp instance as your Context and the framework will automatically localize display strings based on the user who is logged in for that HTTP session.

Note that Workbench code always uses the default locale of the VM, so it is typical to just use `null` for Context. However code designed to run in a station VM should always pass through Context.

Lexicon

Lexicons are Java properties files which store localized key/value pairs. A directory called "file:!lexicon/*lang*" is used to store the lexicon files for a specific language, where lang is the locale code. Within the directory there is a file per module named "*moduleName*.lexicon". Every module with a lexicon should also provide a fallback lexicon bundled in the root directory of module's jar file: "module://*moduleName*/*moduleName*.lexicon" (note in the source tree it is just "module.lexicon").

Access to lexicons is provided via the Lexicon API.

BFormat

Many Niagara APIs make use of the BFormat class to store a formatted display string. BFormat provides the ability to insert special function calls into the display string using the percent sign. One of these calls maps a string defined in a lexicon via the syntax "%lexicon(module:key)%. Whenever a display string is stored as a BFormat, you may store one locale in-place or you may use the %lexicon()% call to indirectly reference a lexicon string.

Slots

One of the first steps in localization, is to provide locale specific slot names. Every slot has a programmatic name and a context sensitive display name. The process for deriving the display name for a slot:

1. **BComplex.getDisplayName(Slot, Context)**: The first step is to call this API. You may override this method to provide your own implementation for localization.
2. **NameMap**: The framework looks for a slot called "displayNames" that stores a BNameMap. If a NameMap is found and it contains an entry for the slot, that is used for the display name. Note the NameMap value is evaluated as a BFormat, so it may contain a lexicon call. NameMaps are useful ways to localize specific slots, localize instances, or to localize dynamic slots.
3. **Lexicon**: Next the framework attempts to find the display name for a slot using the lexicon. The lexicon module is based on the slot's declaring type and the key is the slot name itself.
4. **Slot Default**: If we still haven't found a display name, then we use a fallback mechanism. If the slot is frozen, the display name is the result of `TextUtil.toFriendly(name)`. If the slot is dynamic the display name is the result of `SlotPath.unescape(name)`.

Facets

Sometimes facets are used to store display string. In these cases, the string is interpreted as a BFormat so that a %lexicon()% call may be configured. This design pattern is used for:

- Boolean trueText
- Boolean falseText

FrozenEnums

Compile time enums subclass from BFrozenEnum. Similar to slot names and display names, enums have a programmatic tag and a display tag. Localization of display tags uses the following process:

1. **Lexicon**: The framework first attempts to map the display tag to a lexicon. The module is the declaring type of the FrozenEnum and the key is the programmatic tag.
2. **Default**: If a display tag isn't found in the lexicon, then the fallback is the result of `TextUtil.toFriendly(tag)`.

DynamicEnums

Localization of BDynamicEnums is done via the BEnumRange API. An EnumRange may be associated with a DynamicEnum directly via DynamicEnum.make() or indirectly via Context facets. An EnumRange may be composed of a FrozenEnum's range and/or dynamic ordinal/tag pairs. Any portion of the frozen range uses the same localization process as FrozenEnun. The dynamic portion of the range uses the following process:

1. **Lexicon**: If BEnumRange.getOptions() contains a "lexicon" value, then we attempt to map the display tag to a lexicon where the module is the value of the "lexicon" option and the key is the programmatic tag.
2. **Default**: If a display tag is not found using the lexicon, and the ordinal does map to a programmatic tag, then the result of `SlotPath.unescape(tag)` is returned.
3. **Ordinal**: The display tag for an ordinal that isn't included in the range is the ordinal itself as a decimal integer.

User Interface

When building a user interface via the bajaui APIs, all display text should be localizable via lexicons. In the case of simple BLabels, just using the Lexicon API is the best strategy.

The Command and ToggleCommand APIs also provide built-in support for fetching their label, icon, accelerator, and description from a lexicon. Take the following code example:

```
class DoIt extends Command
{
  DoIt(BWidget owner) { super(owner, lex, "do.it"); }

  static final Lexicon lex = Lexicon.make(MyCommand.class);
}
```

In the example above DoIt would automatically have it's display configured from the declaring module's lexicon:

```
do.it.label=Do It
do.it.icon=module://icons/x16/build.png
do.it.accelerator=Ctrl+D
do.it.description=Do it, whatever it is.
```

Locale Selection

Every time a Niagara VM is started it attempts to select a default locale using the host operating system. The OS default may be overridden via the command line flag "-locale:*lang*", where lang is the locale code. The locale code can be any string that maps to a lexicon directory, but typically it is a ISO 639 locale code such as "fr". The default locale of the VM may be accessed via the `Sys.getLanguage() API.`

When the workbench is launched as a desktop application it follows the rules above to select it's locale. Once selected the entire workbench uses that locale independent of user accounts used to log into stations.

The locale for web browser access to a station follows the rules:

1. **User.language**: If the language property of user is a non-empty string, then it defines the locale to use.
2. **Accept Language**: Next the framework tries to select a locale based on the "Accept-Language" passed in the browser's HTTP request. Typically this is configured in the browser's options.

3. **Default**: If all else fails, then the default locale of the station's VM is used

Time Formatting

The default time format is defined by the lexicon key baja:timeFormat. But it may be selectively overridden by users. To change the time format in the Workbench use General Options under Tools | Options. Use the User.facets property to change it for browser users.

Niagara' time format uses a simple pattern language:

Pattern	Description
YY	Two digit year
YYYY	Four digit year
M	One digit month
MM	Two digit month
MMM	Abbreviated month name
D	One digit day of month
DD	Two digit day of month
h	One digit 12 hour
hh	Two digit 12 hour
H	One digit 24 hour
HH	Two digit 24 hour
mm	Two digit minutes
ss	Seconds (and milliseconds if applicable)
a	AM/PM marker
z	Timezone
anything else	Character literal

In addition to the time format configured by the user, developers may customize the resolution via the following facets:

BFacets.SHOW_TIME

BFacets.SHOW_DATE

BFacets.SHOW_SECONDS

BFacets.SHOW_MILLISECONDS

BFacets.SHOW_TIME_ZONE

To programmatically format a time using this infrastructure use the BAbsTime or BTime APIs.

Unit Conversion

By default the framework displays all numeric values using their configured units (via Context facets). Users may override this behavior to have all values converted to the US/English system or SI/Metric systems. To enable this feature in Workbench use General Options under Tools | Options. Use the User.facets property to enable it for browser users.

The list of units known to the system and how to convert is configured via the file:!lib/units.xml XML file. The mapping of those units between English and Metric is done in the file:!lib/unitConversion.xml XML file.

To programmatically format and auto-convert numerics use the BFloat or BDouble APIs.

Note this unit conversion is independent of the conversion which may be performed by ProxyExts when mapping a point into a driver.

Spy

Overview

The Niagara Framework is built upon a principle of high visibility. By modeling everything as a `BObjects` most data and functionality is automatically made visible using the tools built into the workbench. However it is infeasible to model all data using the component model. The spy framework provides a diagnostics window into the system internals for debugging which goes beyond the component model.

Spy pages are accessed via the spy:/ ord.

See javax.baja.spy package for more details.

Licensing

Overview

The Niagara licensing model is based upon the following elements:

- **HostId** A short String id which uniquely identifies a physical box which runs Niagara. This could a Windows workstation, Jace-NP, or any Jace-XXX embedded platform. You can always check your hostId using the command "nre -version".
- **Certificate:** A file ending in "certificate" which matches a vendor id to a public key. Certificates are granted by Tridium, and digitally signed to prevent tampering. Certificates are stored in the "{home}\certificates" directory.
- **License File:** A file ending in "license" which enables a set of vendor specific features. A licenses file is only valid for a machine which matches its hostId. Licenses are digitally signed by a specific vendor to prevent tampering. License files are stored in the "{home}\licenses" directory.
- **Feature:** A feature is a unique item in the license database keyed by a vendor id and feature name. For example "Tridium:jade" is required to run the Jade tool.
- **API:** The `javax.baja.license` package provides a simple API to perform checks against the license database.

License File

A license file is an XML file with a ".license" extension. License files are placed in "{home}\licenses". The filename itself can be whatever you like, but convention is to name the file based on the file's vendor id. The following is an example license file:

```
<license
     version="1.0"
     vendor="Acme"
     generated="2002-06-01"
     expiration="never"
     hostId="Win-0000-1111-2222-3333">
 <feature name="alpha"/>
 <feature name="beta" expiration="2003-01-15"/>
 <feature name="gamma" count="10"/>
 <signature>MC0CFACwUvUwA+mNXMfogNb6PVURneerAhUAgZnTYb6kBCsvsmC2by1tUe/5k/4=</signature>
</license>
```

Validation

During bootstrap, the Niagara Framework loads its license database based on the files found in the "{home}\licenses" directory. Each license file is validated using the following steps:

1. The `hostId` attribute matches the license file to a specific machine. If this license file is placed onto a machine with a different hostId, then the license is automatically invalidated.
2. The `expiration` attribute in the root element specifies the master expiration. Expiration must be a format of "YYYY-MM-DD". If the current time is past the expiration, the license file is invalidated. The string "never" may be used to indicate no expiration.
3. The `generated` attribute in the root element specifies the license file generation date as "YYYY-MM-DD". If the current time is before the generated date, the license file is invalidated.
4. The `vendor` attribute is used to inform the framework who has digitally signed this license file. In order to use a license file, there must be a corresponding certificate file for that vendor in the "{home}\certificates" directory.
5. The `signature` element contains the digital signature of the license file. The digital signature is created by the vendor using the vendor's private key. The signature is verified against the vendor's public key as found in the vendor's certificate. If the digital signature indicates tampering, the license file is invalid.

Features

A license database is a list of features merged from the machine's license files that are validated using the procedure discussed above.

Each feature is defined using a single XML element called `feature`. Features are identified by the vendor id which is signed into the license file and a feature name defined by the `name` attribute.

The `expiration` attribute may be specified in the feature element to declare a feature level expiration. Expiration is a string in the format of "never" or "YYYY-MM-DD". If expiration is not specified then never is assumed.

Each feature may declare zero or more name/value properties as additional XML attributes. In the example license above the "gamma" feature has one property called "count" with a value of "10".

Predefined Features

The following is a list of predefined features used by the Niagara Framework. All of these features require a vendor id of "Tridium":

- **workbench**: Required to run the workbench tool.
- **station**: Required to run a station database.

API Usage

The following are some snippets of Java code used to access the license database:

```
// verify that the "Acme:CoolFeature" is licensed on this machine
try
{
  Sys.getLicenseManager().checkFeature("Acme", "CoolFeature");
  System.out.println("licensed!");
}
catch(LicenseException e)
{
  System.out.println("not licensed!");
}

// get some feature properties
Feature f = Sys.getLicenseManager().getFeature("Acme", "gamma");
f.check();
String count = f.get("count");
```

Checking Licenses

You may use the following mechanisms to check your license database:

1. Use the console command "nre -licenses".
2. Use the spy page.

XML

Overview

The javax.baja.xml package defines the core XML API used in the Niagara architecture. The two cornerstones of this APIs are:

1. **XElem:** Provides a standard representation of an XML element tree to be used in memory. It is similar to the W3's DOM, but much lighter weight.
2. **XParser:** XParser is a light weight XML parser. It may be used in two modes: to read an entire XML document into memory or as a pull-parser.

The Baja XML APIs are designed to be small, fast, and easy to use. To achieve this simplicity many advanced features of XML are not supported by the `javax.baja.xml` APIs:

- Only UTF-8 and UTF-16 encodings are supported. Unicode characters in attributes and text sections are escaped using the standard entity syntax '&#dd;' or '&#xhh;'.
- All element, attribute, and character data productions are supported.
- CDATA sections are supported.
- Namespaces are supported at both the element and attribute level.
- Doctype declarations, DTDs, entity declarations are all ignored by the XML parser. XML used in Niagara is always validated at the application level for completeness and efficiency.
- Processing instructions are ignored by the XML parser.
- No access to comments is provided by the XML parser.
- Character data consisting only of whitespace is always ignored.

Example XML

For the code examples provided we will use this file "test.xml":

```
<root xmlns="ns-stuff" xmlns:u="ns-user">
  <u:user name="biff" age="29">
    <u:description>Biff rocks</u:description>
    <u:skills sing="true" dance="false"/>
  </u:user>
  <user name="elvis" alive="maybe" xmlns="">
    <skills sing="true" dance="true"/>
  </user>
  <attr1="1" u:attr2="2"/>
</root>
```

Working with XElem

The XElem class is used to model an XML element tree. An element is defined by:

- **Namespace:** Elements which are in a namespace will return a non-null value for `ns()`. You may also use the `prefix()` and `uri()` methods to access the namespace prefix and URI. The "xmlns" attribute defines the default namespace which will apply to all child elements without an explicit prefix. The "xmlns:{prefix}" attribute defines an namespace used by child elements with the specified prefix.
- **Name:** The `name()` method returns the local name of the element without the prefix. You may also use `qname()` to get the qualified name with the prefix.
- **Attributes:** Every element has zero or more attributes declared within the element start tag. There are an abundance of convenience methods used to access these attributes. Attributes without an explicit prefix are assumed to be in no namespace, not the default namespace.
- **Content:** Every element has zero of more content children. Each content child is either an XText or XElem instance.

Character data (including CDATA) is represented using XText.

The following code illustrates many of the commonly used methods on XElem:

```
// parse the test file
XElem root = XParser.make(new File("test.xml")).parse();

// dump xml tree to standard out
root.dump();

// dump root identity
System.out.println("root.name   = " + root.name());
System.out.println("root.ns     = " + root.ns());

// get elements
System.out.println("elems()     = " + root.elems().length);
System.out.println("elems(user) = " + root.elems("user").length);

// biff
XElem biff = root.elem(0);
System.out.println("biff.name   = " + biff.name());
System.out.println("biff.ns     = " + biff.ns());
System.out.println("biff.age    = " + biff.get("age"));

// elvin
XElem elvis = root.elem(1);
XElem skills = elvis.elem("skills");
System.out.println("elvis.name  = " + elvis.name());
System.out.println("elvis.ns    = " + elvis.ns());
System.out.println("skills.sing = " + skills.getb("sing"));
```

Output from code above:

```
root.name   = root
root.ns     = ns-stuff
elems()     = 3
elems(user) = 2
biff.name   = user
biff.ns     = ns-user
biff.age    = 29
elvis.name  = user
elvis.ns    = null
skills.sing = true
```

Working with XParser

The XParser class is used to parse XML input streams into XElems. The easiest way to do this is to parse the entire document into memory using the `parse()` method:

```
// parse and close input stream
XElem root = XParser.make(in).parse();
```

The above code follows the W3 DOM model of parsing a document entirely into memory. In most cases this is usually acceptable. However it can create efficiency problems when parsing large documents, especially when mapping the XElems into other data

structures. To support more efficient parsing of XML streams, XParser may also be used to read elements off the input stream one at a time. This is similar to the SAX API, except you pull events instead of having them pushed to you. A pull model is much easier to work with.

To work with the pull XParser APIs you will use the `next()` method to iterate through the content instances. This effectively tokenizes the stream into XElem and XText chunks. Each call to `next()` advances to the next token and returns an int constant: `ELEM_START`, `ELEM_END`, `TEXT`, or `EOF`. You may also check the type of the current token using `type()`. You may access the current token using `elem()` or `text()`.

XParser maintains a stack of XElems for you from the root element down to the current element. You may check the depth of the stack using the `depth()` method. You can also get the current element at any position in the stack using `elem(int depth)`.

It is very important to understand the XElem at given depth is only valid until the parser returns ELEM_END for that depth. After that the element will be reused. The XText instance is only valid until the next call to `next()`. You can make a safe copy of the current token using `copy()`.

The following code illustrates using XParser in pull mode:

```
XParser p = XParser.make(new File("test.xml"));

p.next(); // /root start
System.out.println("root.start: " + p.elem().name() + " " + p.depth());
p.next(); // root/user(biff) start
System.out.println("biff.start: " + p.elem().name() + " " + p.depth());
p.next(); // root/user/description start
System.out.println("desc.start: " + p.elem().name() + " " + p.depth());
p.next(); // root/user/description text
System.out.println("desc.text:  " + p.text() + " " + p.depth());
p.next(); // root/user/description end
System.out.println("desc.end:   " + p.elem().name() + " " + p.depth());
p.skip(); // skip root/user/skills
p.next(); // root/user(biff) end
System.out.println("biff.end: " + p.elem().name() + " " + p.depth());
p.next(); // root/user(elvis) start
System.out.println("elvis.start: " + p.elem().name() + " " + p.depth());
```

Output from code above:

```
root.start:  root        1
biff.start:  user        2
desc.start:  description 3
desc.text:   Biff rocks  3
desc.end:    description 3
biff.end:    user        2
elvis.start: user        2
```

Bog Files

Overview

Niagara provides a standard XML format to store a tree of BValues. This XML format is called "bog" for Baja Object Graph. The bog format is designed for the following criteria:

- Easy to serialize a graph to an output stream using one pass;
- Easy to deserialize a graph from an input stream using one pass;
- Compact XML, using single letter element and attribute names;
- Ability to compress using zip;

Bog files are typically given a ".bog" extention. Although the ".palette" extension can be used to distinguish a bog designed for use as palette; other than extension bog and palette files are identical.

Bog files can be flat XML files or stored inside zip files. If zipped, then the zip file contains a single entry called "file.xml" with the XML document. You use workbench to copy any BComponent to a directory on your file system to easily generate a bog.

API

In general the best way to read and write bog files is via the standard APIs. The `BogEncoder` class is used to write BValues to an output stream using bog format. Note that `BogEncoder` subclasses `XWriter` for generating an XML document. You can use the `XWriter.setZipped()` method to compress the bog file to to a zip file with one entry called "file.xml". In general you should use the `encodeDocument()` method to generate a complete bog document. However you can also use `BogEncoder` to stream multiple BValues to an XML document using `encode()`.

The `BogDecoder` class is used to decode a bog document back into BValue instances. Note that `BogDecoder` subclasses `XParser` for parsing XML. When decoding a bog file, `XParser` will automatically detect if the file is zipped or not. General usage is to use `decodeDocument()` in conjunction with `BogEncoder.encodeDocument()` for decoding the entire XML document as a BValue. However `BogDecoder` can also be used to decode BValues mixed with other XML data using `BogDecoder.decode()` and the standard `XParser` APIs.

`BogEncoder.marshal()` and `BogDecoder.unmarshal()` are convenience methods to encode and decode a BValue to and from a String.

Syntax

The bog format conforms to a very simple syntax. The root of a bog document must always be "bajaObjectGraph". Under the root there are only three element types, which map to the three slot types:

Element	Description
p	Contains information about a property slot
a	Contains information about a frozen action slot
t	Contains information about a frozen topic slot

All other information is encoded into XML attributes:

Attribute	Description
n	This required attribute stores the slot name.
m	Defines a module symbol using the format "symbol=name". Once defined, the symbol is used in subsequent t attributes.
t	Specifies the type of a property using the format "symbol:typename", where symbol must map to a module declaration earlier in the document. If unspecified, then the type of the property's default value is used.

f	Specifies slot flags using the format defined by `Flags.encodeToString()`
h	This attribute specifies the handle of BComponents
x	Specifies the slot facets using format defined by `BFacets.encodeToString()`
v	Stores the string encoding of BSimples.

In practice the XML will be a series of nested `p` elements which map to the structure of the BComplex tree. The leaves of tree will be the BSimples stored in the `v` attribute.

Example

A short example of a kitControl:SineWave linked to a kitControl:Add component. The Add component has a dynamic slot called description where value is "hello", operator flag is set, and facets are defined with multiLine=true.

```
<?xml version="1.0" encoding="UTF-8"?>
<bajaObjectGraph version="1.0">
<p m="b=baja" t="b:UnrestrictedFolder">
 <p n="SineWave" h="1" m="kitControl=kitControl" t="kitControl:SineWave">
  </p>
  <p n="amplitude" v="35"/>
 </p>
 <p n="Add" h="3" t="kitControl:Add">
  </p>
  <p n="Link" t="b:Link">
   <p n="sourceOrd" v="h:1"/>
   <p n="sourceSlotName" v="out"/>
   <p n="targetSlotName" v="inA"/>
  </p>
  <p n="description" f="o" x="multiLine=b:true" t="b:String" v="hello"/>
 </p>
</p>
</bajaObjectGraph>
```

Distributions

Overview

A distribution is a platform-specific archive of deployable software. The distribution file:

- Is a JAR file compliant with PKZIP compression;
- Contains an XML manifest in meta-inf/dist.xml;
- Contains files to be deployed;
- States its dependencies on any *parts* such as hardware, operating system, Niagara or third-party software

JAR Entry Paths

The JAR entry paths mirror their intended filesystem paths directly. Paths are mapped either to the target host's root directory or the {baja home} directory, according to the manifest. Unless specified otherwise by the manifest or end-user request, and except for the dist.xml manifest itself, each file will be copied to the target host (*i.e.*, there is no facility provided for the user or an installation program to choose which specific pieces to install).

Manifest

The distribution manifest is found in the meta-inf/dist.xml JAR entry. It

- provides some high-level descriptive information about the distribution,
- specifies the distribution's external dependencies and exclusions,
- provides a summary description of its contents to assist installer software in dependency analysis,
- identifies modifications that need to be made to the host's platform.bog file, and
- specifies under which conditions existing files are to be replaced.

An example distribution file manifest is provided as a reference for the remaining specification:

```
<dist name="qnx-jace-york"
      version="2.1.6"
      description=""
      buildDate="Thu Jan 18 10:58:39 Eastern Standard Time 2007"
      buildHost="BRUTUS"
      reboot="true"
      noRunningApp="true"
      absoluteElementPaths="true"
      osInstall="true"
>


<dependencies>
  <part name="york" desc="York System Board" />
</dependencies>

<exclusions>
  <os name="qnx-jace-york" version="2.2" />
</exclusionss>

<provides>
  <os name="qnx-jace-york" version="2.1.6" />
</provides>
<fileHandling>
  <file name="dev/shmem/york.image" replace="oscrc"/>
```

```
</fileHandling>
</dist>
```

The root **dist** element has the following attributes:

- **name** [required]: Name of the distribution.
- **version** [required]: Version of the distribution, dot delimited. If version is nonzero, then it and the name must uniquely identify the file's contents - they cannot be reused with different contents.
- **vendor** [optional]: Vendor name for the provider of the distribution.
- **description** [optional]: Brief description of the what the distribution provides.
- **buildDate** [optional]: Timestamp for when the distribution was created. May be generated by a build tool, useful in tracking development builds.
- **buildHost** [optional]: Host on which the distribution was created. May be generated by a build tool, useful in tracking development builds.
- **reboot** [optional]: true or false, defaults to true. If true, the host to which this distribution is installed must be rebooted after installation is complete.
- **noRunningApp** [optional]: true or false, defaults to true. If true, then the distribution may not be installed while any Niagara stations or Sedona applications are running.
- **absoluteElementPaths** [optional]: true or false, defaults to false. If true, then the JAR entry path maps directly to the host's root directory, otherwise the entry path maps to {baja_home}.
- **osInstall** [optional]: true or false, defaults to false. Meaningless for Win32 platforms. If true, then the distribution contains an operating system image that must be installed after the files are downloaded and before the host is rebooted.

platform Element [optional]

Sub-elements are XML in bog file format and will be merged *by name* into the host's existing platform.bog file, or will be used to initialize a new platform.bog file. This allows a distribution to provide new platform services without clobbering or re-initializing any existing platform services.

dependencies Element [optional]

describes the *parts* which must be present on a target host before the distribution can be installed. Each *part* is identified by a sub-element:

arch element [optional]

Describes a chipset dependency. Required **name** attribute specifies the chipset architecture name.

model element [optional]

Describes a model dependency. Required **name** attribute specifies the model name.

os element [optional]

Describes an operating system dependency. Attributes:

- **name** [required]: operating system name
- **vendor** [optional]: operating system vendor
- **version** [required]: operating system version, dot delimited.
- **rel** [optional]: describes how the version is evaluated. Possible values are "minimum" (default), "maximum", and "exact".

nre element [optional]

Describes a dependency on a Niagara runtime version. Attributes:

- **name** [required]: NRE name
- **vendor** [optional]: NRE vendor
- **version** [required]: NRE version, dot delimited.
- **rel** [optional]: describes how the version is evaluated. Possible values are "minimum" (default), "maximum", and "exact".

brand element [optional]

Describes a dependency on a brand. Required attribute **name** uniquely identifies the brand that must be specified in a target host's license for the dependency to be met.

module element [optional]

Describes a module dependency. Attributes are the same as those for the **dependency** element in the module.xml manifest (see modules.html), and the optional **rel** option is supported.

part element [optional]

Describes a dependency on any other kind of *part*, such as a piece of hardware. Required attribute **name** is a unique name for the part, and required attribute **description** provides a brief description. Optional **version** attribute, if present, specifies a version requirement for the part, and the optional **rel** attributed specifies how the version is evaluated.

exclusions Element [optional]

Describes the *parts* which must *not* be present on a target host if the distribution is to be installed. The sub-elements allowed are the same as those used in the **dependencies** section.

provides Element [optional]

If present, describes exactly one **part** that is to be installed by this distribition, whose part name matches the distribution name (*i.e.* if the part is named qnx-jace-james, the dist file must be named qnx-jace-james.dist).

If the distribution file doesn't contain a single discrete named, versioned part (for example a system backup) then it must omit the provides element. Also, if for any other reason the distribution should not be installed automatically by an client to satisfy dependencies expressed in other files, it must omit the provides element.

The part is described by a sub-element:

os element [optional]

Describes an operating system element. Attributes:

- **name** [required]: operating system name
- **vendor** [optional]: operating system vendor
- **version** [required]: operating system version, dot delimited.

vm element [optional]

Describes an installable java virtual machine. Attributes:

- **name** [required]: virtual machine name
- **vendor** [optional]: VM vendor
- **version** [required]: VM version, dot delimited.

nre element [optional]

Describes an installable Niagara runtime engine (NRE). Attributes:

- **name** [required]: NRE product name
- **vendor** [optional]: NRE vendor
- **version** [required]: NRE version, dot delimited.

fileHandling Element [optional]

Specifies the rules by which files are replaced, and identifies directories and files which must be removed before installation begins. The optional **replace** attribute specifies the rules for replacing existing files - its values can be "always" if the file is always to be replaced, "never" if it is never to be replaced, and "crc" if the file is to be replaced only if the CRC checksums for the distribution and current versions of the file are not the same. By default, no files or directories are to be removed by the installer prior to installation, and the "crc" replacement rule is used. The **fileHandling** element may contain the following sub-elements:

remove element

Specifies a file or directory to be removed prior to installation. Its required **name** attribute specifies the path (according to

absoluteElementPaths element) to the file/directory. If **name** specifies a directory, exceptions may be specified using the **keep** sub-element.

keep element [optional]

Specifies a file or directory that should not be removed as the result of a **remove** element. Its required **name** attribute specifies the path.

file element [optional]

Specifies a file replacement rule that differs from the default in the **fileHandling** element or any **dir** element that might apply to the file's path. Its required **name** attribute specifies the file path, and the required **replace** attribute specifies the rules for replacing the existing file - its values can be "always" if the file is always to be replaced, "never" if it is never to be replaced, "crc" if the file is to be replaced only if the CRC checksums for the distribution and current versions of the file are not the same, and "oscrc" if the file's CRC checksum is to be checked against a CRC value returned by the niagarad for the OS image.

dir element [optional]

Specifies a file replacement rule for a given directory path that differs from the default in the **fileHandling** element or in **dir** elements for parent paths. Its required **name** attribute specifies the directory's path. Its optional **replace** attribute can be "always", "never", or "crc". Its optional **clean** attribute, if present and equal to "true", indicates that any file or subdirectory that isn't part of the distribution should be deleted by the installer.

Test

Overview

Niagara includes a unit testing framework very similiar to JUnit, in the package `javax.baja.test`. There are two key pieces of the test framework: `BTest` is the base class for defining new test cases and `TestRunner` is used to run a suite of tests.

- ## BTests

The `BTest` class is the base class used to define new test cases. Create subclasses of `BTest` to create new tests. Any public methods on your test class which begin with `"test"` will be your test methods. Within your test methods, you write code to assert a series of conditions by calling the `BTest.verifyXXX()` methods.

A fresh instance of your test class is created for each test method invocation. Your class will receive the `setup()` callback before each test method and the `cleanup()` callback after. The lifecycle of a test run:

1. Create new instance of test class
2. Call setup()
3. Call testXXX() method
4. Call cleanup()
5. Repeat 1. for all test methods

BTest Example

Here is a simple example to test `java.lang.String`:

```
// acmeStuff:StringTest
public class BStringTest
  extends BTest
{

  public Type getType() { return TYPE; }
  public static final Type TYPE = Sys.loadType(BStringTest.class);

  public void testCharAt()
  {
    verifyEq("abc".charAt(0), 'a');
    verifyEq("abc".charAt(1), 'b');
    verifyEq("abc".charAt(2), 'c');
  }

  public void testTrim()
  {
    verifySame("abc".trim(), "abc");
    verifyEq(" abc".trim(), "abc");
    verifyEq("\nabc\t ".trim(), "abc");
  }

  public void testSubstring()
  {
    verifyEq("hello".substring(0, 1), "h");
    verifyEq("hello".substring(2, 4), "ll");
    verifyEq("hello".substring(1), "ello");
```

```
  }
}
```

The example above has three test methods: `testCharAt()`, `testTrim()`, and `testSubstring()`. Each test asserts a series of conditions - in this case using either `verifyEq` or `verifySame`.

TestRunner

The `TestRunner` class manages running `BTests`. You can use the `TestRunner` API directly or subclass it to plug the Niagara test framework into other tools. But the easiest way to run tests is from the command line using `test.exe` which wraps `TestRunner.main(String[])`. The following illustrate some different ways to run the test suite:

```
// run all tests found in the registry:
test all

// run all tests found in acmeStuff module:
test acmeStuff

// run all test methods found on acmeStuff:StringTest type:
test acmeStuff:StringTest

// run just the testTrim() method of StringTest:
test acmeStuff:StringTest.testTrim

// run built-in framework tests found in the "test" module:
test test
```

Virtual Components

Overview

Refer to the Virtual API (available only in Niagara 3.2 and beyond).

The virtual components feature was originally driven by a common use case of most drivers in Niagara AX. However, since the original brainstorming for "phantom" components (later termed "virtual" components), it has since grown to cover a broader range of possible applications. This document (intended for developers) will focus its examples on driver applications, but the idea of transient, on-demand components can obviously reach to many other applications.

As mentioned, the term virtual components refers to transient, on-demand components in a station database that only exist when needed. Virtual components are created dynamically only when they are first required by the station (ie. enter a subscribed state), and then when they are no longer needed (ie. enter an unsubscribed state), they are automatically cleaned up from the station database (subject to virtual cache life constraints). This lifecycle for virtual components provides for efficiency. The key concepts that drive virtual components are their virtual Ords (Object Resolution Descriptors) and their existence within a virtual component space. The Ords for virtual components follow the SlotPath design (refer to VirtualPath) and must uniquely define virtual components (and provide enough information to create the virtual component at runtime). These unique, on-demand virtual components live within a Virtual Component Space, which is different from the normal component space which manages components that are persisted in the station. The link between the normal component space and the virtual component space is through the Virtual Gateway. There is a one-to-one relationship between a virtual gateway and its corresponding virtual component space, so it is possible to have multiple virtual gateways and virtual component spaces in the same running station. These concepts will be described in more detail in the class descriptions that follow.

From a drivers perspective, virtual components means that driver data can be addressed without premapping. Prior to this new feature, the old Niagara AX model used by drivers boiled down to a collection of BComponents used to normalize driver data. For example, most drivers contain a device network, devices, and proxy points (control points with proxy extensions). Proxy points are useful for modeling the smallest pieces of driver data ("point" information) and normalizing them for use in the Niagara AX environment. This model works well for linking proxy points to control logic for monitor and control. The problem with this model is that every piece of driver data that a user may want to visualize/configure in Niagara AX requires the overhead of a persistent component (i.e. proxy point) existing somewhere in the station's component space. The overhead of having persistent, premapped components limits the capacity of points that a station can monitor. This limitation especially becomes a problem on small embedded platforms (such as a JACE) where memory is limited.

There are two common driver use cases we identified where a user might want to have access to driver data, while not wanting the extra overhead of using persistent components. The first is that a user wants to build a Px view to look at device point data (simply for monitoring purposes). In this case, simply a polled value is sufficient to present the data to the user in the view only when it is needed (the view is open). The second use case is for configuration/commissioning a device in which the user wants to see a snapshot (i.e. property sheet) of the values within the device, and allow the user to monitor/modify these device values for one time configuration purposes. In both of these cases, building persistent components to model the driver data is not necessary and simply costs the user extra overhead. Instead, a transient display of the driver data is useful only when the user enters the view, but at all other times, the values are not needed and do not need to be consuming memory (i.e. not needed for linking to any other logic). Thus virtual components is a solution to both of these use cases.

In general, linking in the virtual component space is not supported, as virtual components are not persisted (thus any user created links would be lost).

The Virtual API

The javax.baja.virtual package contains the base classes for supporting virtuals. The following gives a brief description of each class in this package:

BVirtualComponent

A `BVirtualComponent` is a BComponent, however it extends the functionality to support living in a virtual component space by keeping track of its last active ticks. The last active ticks are the clock ticks when the virtual component was last needed (ie. the moment the virtual component switches from a subscribed state back to an unsubscribed state, the last active ticks are updated to

indicate the ticks when the virtual component was last in use*). The last active ticks for each virtual component in the virtual component space are consistently monitored by the space's `VirtualCacheCallbacks` instance, which uses this information to determine when the virtual component is subject to auto-removal (clearing from the cache). Virtual Components can also be spared from auto removal if the instance is the root component of the virtual component space, or if the auto-removal behavior is specifically disabled for the virtual component (by subclassing and overridding the `performAutoRemoval()` callback). By default, virtual components also override the normal `BComponent` behavior to specify their virtual nav Ord, enforce a few parent/child restrictions**, and provide a convenient way to retrieve the parent `BVirtualGateway` instance, which is important because the gateway is the link between the normal component space and the virtual component space.

The `BVirtualComponent` class is the key structure to use for modeling objects in your virtual space. You can use `BVirtualComponent` (or a subclass of it) to model your data (or data groupings), and since `BVirtualComponent` is itself a `BComponent`, it supports all of the normal component life-cycle features. Just remember BComponent instances (those that aren't BVirtualComponents or BVectors) should not be used in the virtual component space, so keep this in mind when determining what types of frozen/dynamic slots your BVirtualComponents need to model the data.

* **NOTE** - The last active ticks for a virtual component are also modified by a "touch" feature of the navigation tree in Workbench. Basically, for any virtual component's nav tree node currently in view in Workbench, there is a periodic message sent that "touches" the virtual component, in order to keep it active and prevent it from being auto cleaned. This is useful because a virtual component simply viewed in the nav tree is not guaranteed to be in a subscribed state.

** **NOTE** - The general rule that should be followed is that the virtual component space should not contain `BComponent` instances that are not `BVirtualComponent`s. So there are a few child/parent checks in place that attempt to enforce this rule. Of course, `BVirtualComponent` instances living within a virtual component space can contain other non-`BComponent` children, such as `BSimple`s, `BStruct`s, and there is even an exception made for `BVector`s. The reason you should keep non-virtual `BComponent` children out of the virtual component space is because it can break the virtual cache cleanup mechanism (discussed below for the `VirtualCacheCallbacks` class).

BVirtualComponentSpace

The `BVirtualComponentSpace` is an extension of BComponentSpace which contains a mapping of `BVirtualComponents` (organized as a tree). The virtual component space is created at runtime when a `BVirtualGateway` instance is started in the station's component space. There is a one-to-one relationship between the virtual gateway and its virtual component space. The virtual component space has a few supporting *Callbacks* classes. In addition to those provided by `BComponentSpace` (`LoadCallbacks`, `SubscribeCallbacks`, and `TrapCallbacks`), `BVirtualComponentSpace` kicks off an instance of `VirtualCacheCallbacks` (described below). It is important to remember that the scope of the virtual component space is limited to its tree of virtual components, but it also has a reference to its `BVirtualGateway` instance which provides the link to the normal component space.

BVirtualGateway

`BVirtualGateway` is an abstract subclass of `BComponent` designed to reside in the station component space and act as a "gateway" to its corresponding virtual component space. As mentioned previously, there is a one-to-one relationship between the virtual gateway and its virtual component space. For the virtual gateway, this means that the nav children displayed under the gateway in the nav tree will be the nav children for the root component of the virtual space. Just to clarify the point, the virtual gateway functions as the link between the normal (station) component space and its virtual space. Thus it overrides all of the nav methods to route to the virtual space's tree (of virtual components). In practice, you should *always avoid* adding frozen/dynamic slots as children of the virtual gateway directly, as the nav overrides will route users to the virtual space by default, thus making it difficult and confusing to view/change slots that are direct children on the virtual gateway itself.

The other important function of the virtual gateway is to provide the hooks for subclasses to load/create virtual components at runtime. This includes a few callback methods that the framework makes to the virtual gateway to tell it to load an individual virtual slot or load all of the virtual slots for a given virtual component. Two important factors to consider when subclassing `BVirtualGateway` and its methods are:

- By contract, whenever slots are added to virtual components, they should always be assigned a slot name that is the *escaped* virtual path name (ie. use `SlotPath.escape(virtualPathName)`). This is very important as virtual path names can be unescaped, but the contract is that their corresponding slot path name is simply the escaped version of the virtual path name. In order for virtual lookup to work correctly, this rule must be followed.
- Due to the possibility of a partial loaded state supported by virtuals, when you subclass `BVirtualGateway` (and even

BVirtualComponent) and implement its methods, you should always be keenly aware of the present subscription state of the virtual components. For example, the BVirtualGateway load methods could be called and cause a new slot to be created for a parent virtual component while that parent is already in a subscribed state. So this could affect how the new virtual slot should be handled (ie. it may need to be added to a poll scheduler for updates). Subclasses should always be aware of this potential state and perform the proper checks to handle this case.

BVirtualScheme

BVirtualScheme extends BSlotScheme and defines the "virtual" ord scheme ID. It works in close conjunction with VirtualPath for resolving virtual Ords (see below for further details).

VirtualCacheCallbacks

This class is instantiated by BVirtualComponentSpace with a purpose to manage the virtual cache (ie. to determine when its appropriate to auto cleanup virtuals that are no longer in use). The default implementation of VirtualCacheCallbacks has a shared thread pool (used by multiple virtual component space instances) designed to monitor the virtual components in each registered virtual space, and check the min/max virtual cache life for any unused virtual components. The idea is that virtual components, when no longer needed, will remain in the cache for a certain cache life before they get automatically removed. The following static variables allow for tuning the performance of the virtual cache management (**all default values can be tweaked by making the appropriate settings in the system.properties file**):

public static final BRelTime MAX_CACHE_LIFE - Specifies the default virtual cache life maximum (default 45 seconds). When a virtual component expires, it will remain in memory for a maximum of this amount of time before it will be automatically cleaned up from the cache (assuming the virtual component is not re-activated in the meantime).

public static final BRelTime MIN_CACHE_LIFE - Specifies the default virtual cache life minimum (default 25 seconds). When a virtual component expires, it will remain in memory for a minimum of this amount of time before it will be subject to automatic clean up from the cache (assuming the virtual component is not re-activated in the meantime). This minimum cache life is only a factor when the virtual threshold limit has been exceeded (meaning that virtuals need to be cleaned up faster than normal). If the virtual threshold limit has not been exceeded, the maximum virtual cache life will be used (normal operation).

public static final int VIRTUAL_THRESHOLD - Specifies a global virtual threshold limit (default 1000), above which virtuals will start being auto cleaned from the cache quicker as space is needed (ie. the MIN_CACHE_LIFE will be used in the cache life determination when the number of virtuals in the station exceeds this threshold limit, otherwise the MAX_CACHE_LIFE will be used when the number of virtuals doesn't exceed this limit).

public static final long VIRTUAL_THRESHOLD_SCAN_RATE - Specifies the default time (in milliseconds) in which to perform a full scan of the station for virtuals, used for threshold level checking. The default is 1000, which means that every second, a full scan will occur. A value of zero disables the virtual threshold checking feature entirely.

public static final int THREAD_POOL_SIZE - Specifies the maximum number of worker threads in the thread pool shared by VirtualCacheCallbacks instances. There is a VirtualCacheCallbacks instance per virtual component space, however, the default implementation shares a common worker thread pool. Therefore, this setting determines the maximum number of virtual cleanup worker threads (10 default).

public static final int SPACES_PER_THREAD - Specifies the ideal number of virtual component spaces managed per worker thread in thread pool (this limit can be exceeded if all threads in the pool are already at capacity). The default is 5 virtual spaces (optimum) per thread.

VirtualPath

VirtualPath extends SlotPath and allows for resolving BVirtualComponents (and their child slots) using unescaped slot names in the path (note that this is different from SlotPath which enforces the rule that only escaped slot names can be contained in the path). The '/', '|', '$', and ':' characters are reserved and not allowed in a virtual path entry. Also, the "../" is reserved for backups.

The most common use case of VirtualPath follows the following format:

```
host:|session:|space:|Ord to virtual gateway:|virtual:virtual path
```

For example (disregard the line wrap):

```
local:|fox:|station:|slot:/Config/Drivers/YourNetwork/YourDevice/YourVirtualGateway
|virtual:/Virtual Component A/Virtual Component B/Output Value
```

This example shows how the virtual gateway is always the link point between the normal component space and the virtual space. The "|virtual:" in the example above indicates the jump to the virtual component space. When resolving such an Ord, once it starts parsing the virtual path, it will start from the left and work to the right (the "/" acts as the separator between virtual slots). So this means it will first check for the existence of a slot named "Virtual$20Component$20A" under the root component of the virtual space and return it if it exists (remember that by contract, virtual path names should be escaped to form the slot name). If it doesn't already exist, the virtual gateway will be given the opportunity to create a virtual object to represent it given the virtual path name and parent (subclasses will normally put enough information in the virtual path to know how to create the object). This process continues from left to right until the virtual path has resolved the last in the list. The example above would be represented in the nav tree like this:

```
Config
|
|
|____Drivers
     |
     |
     |____YourNetwork
          |
          |
          |____YourDevice
               |
               |
               |____YourVirtualGateway
                    | (entrance to virtual space)
                    | virtual space root component (hidden)
                    |____Virtual Component A
                         |
                         |
                         |____Virtual Component B
                              |
                              |
                              |____Output Value
```

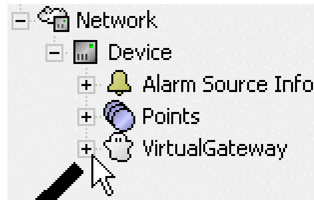
It is also worth noting that due to the virtual/slot path name contract, the following Ord is ***functionally equivalent*** to the example above (disregard the line wrap):

```
local:|fox:|station:|slot:/Config/Drivers/YourNetwork/YourDevice/YourVirtualGateway
|virtual:|slot:/Virtual$20Component$20A/Virtual$20Component$20B/Output$20Value
```

Virtual Component Lifecycle

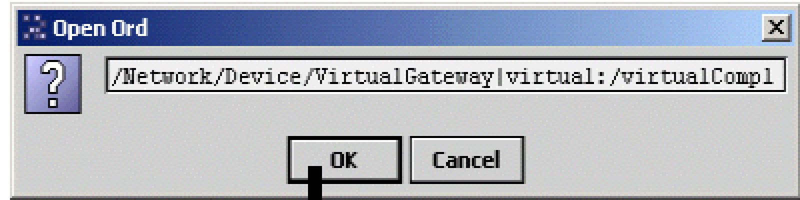
The following diagram attempts to show the common lifecycle of a virtual component.

User expands BVirtualGateway in the Nav tree…

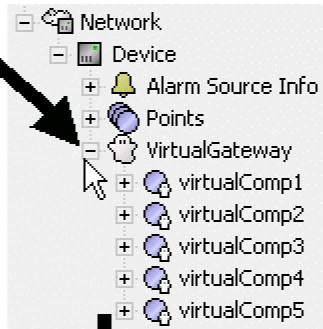


OR

A Virtual Ord is opened (ie. due to a Virtual Ord binding in an opened Px view), leading to the Virtual Ord resolution…



If not fully loaded already, calls BVirtualGateway.loadVirtualSlots() to discover BVirtualComponent children



Calls BVirtualGateway.loadVirtualSlot()

BVirtualComponent already exists

BVirtualComponent doesn't exist

Calls BVirtualGateway.addVirtualSlot() to create the BVirtualComponent

If BVirtualComponent opened in a view (ie. subscribed)

unsubscribe()

If BVirtualComponent only visible in Nav tree

VirtualCacheCallbacks monitors all BVirtual Components checking for expiration (inactive for cache life)

BVirtualComponents visible in the Nav tree receive touch() to stay active

BVirtualComponent expires

Gx Graphics Toolkit

Overview

The gx module defines the graphics primitives used for rendering to a display device. For example there implements for "painting" to computer screens and another for "painting" a PDF file. Many of the simple types used in the rest of the stack are defined in gx including BColor, BFont, BPen, BBrush, BGeom, and BTransform.

The gx APIs use a vector coordinate system based on x and y represented as doubles. The origin 0,0 is the top left hand corner with x incrementing to the right and y incrementing down. This coordinate system is called the logical coordinate space (sometimes called the user space). How the logical coordinate space maps to the device coordinate space is environment specific. Usually a logical coordinate maps directly into pixels, although transforms may alter this mapping.

Color

BColor stores an RGBA color. It's string syntax supports a wide range of formats including most specified by CSS3:

- **SVG Keywords**: the full list of X11/SVG keywords is available by name and also defined as constants on BColor. Examples: red, springgreen, navajowhite.
- **RGB Function**: the rgb() function may be used with the red, green, and blue components specified as an integer between 0-255 or as a percent 0%-100%. Examples: rgb(0,255,0), rgb(0%,100%,0%).
- **RGBA Function**: the rgba() function works just like rgb(), but adds a fourth alpha component expressed as a float between 0.0 and 1.0. Example: rgba(0,100,255,0.5).
- **Hash**: the following hash formats are supported #rgb, #rrggbb, and #aarrggbb. The first two follow CSS rules, and the last defines the alpha component using the highest 8 bits. Examples: #0b7, #00bb77, #ff00bb77 (all are equivalent).

Font

BFont is composed of three components:

- **Name**: a font family name stored as a String.
- **Size**: a point size stored as a double.
- **Style**: a set of attributes including bold, italics, and underline.

The format of fonts is "[italic || bold || underline] {size}pt {name}". Examples: "12pt Times New Roman", "bold 11pt sans-serif", "italic underline 10pt Arial". The `BFont` class also provides access to a font's metrics such as baseline, height, ascent, descent, and for calculating character widths.

Brush

The BBrush class encapsulates how a shape is filled. The gx brush model is based on the SVG paint model. There are four types of brushes:

- **Solid Color**: the string format is just a standard color string such as "red"
- **Inverse**: uses an XOR painting mode
- **Gradients**: linear and radial gradients
- **Image**: based on a bitmap image file which may tiled or untiled

Pen

The BPen class models how a geometric shape is outlined. A pen is composed of:

- **Width**: as double in logical coordinates
- **Cap**: how a pen is terminated on open subpaths - capButt, capRound, capSquare.
- **Join**: how corners are drawn - joinMiter, joinRound, joinBevel
- **Dash**: a pattern of doubles for on/off lengths

Coordinates

The following set of classes is designed to work with the gx coordinate system. Each concept is modeled by three classes: an interface, a mutable class, and an immutable BSimple version that manages the string encoding.

Point

The IPoint, Point, and BPoint classes store an x and y location using two doubles. The string format is "x, y". Example "40,20", "0.4,17.33".

Size

The ISize, Size, and BSize classes store a width and height using two doubles. The string format is "width,height". Examples include "100,20", "10.5, 0.5".

Insets

The IInsets, Insets, and BInsets classes store side distances using four doubles (top, right, bottom, and right). The string format for insets follows CSS margin style: "top,right,bottom,left". If only one value is provided it applies to all four sides. If two values are provided the first is top/bottom and the second right/left. If three values are provided the first is top, second is right/left, and third is bottom. Four values apply to top, right, bottom, left respectively. Examples "4" expands to "4,4,4,4"; "2,3" expands to "2,3,2,3"; "1,2,3" expands to "1,2,3,2".

Geom

The geometry classes are used to model rendering primitives. They all following the pattern used with Point, Size, and Insets with an interface, mutable class, and immutable BSimple. Geometries can be used to stroke outlines, fill shapes, or set clip bounds.

Geom

The IGeom, Geom, and BGeom classes are all abstract base classes for the geometry APIs.

LineGeom

The ILineGeom, LineGeom, and BLineGeom classes model a line between two points in the logical coordinate system. The string format of line is "x1,y1,x2,y2".

RectGeom

The IRectGeom, RectGeom, and BRectGeom classes model a rectangle in the logical coordinate system. The string format of rectangle is "x,y,width,height".

EllipseGeom

The IEllipseGeom, EllipseGeom, and BEllipseGeom classes model a ellipse bounded by a rectangle in the logical coordinate space. The string format is "x,y,width,height".

PolygonGeom

The IPolygonGeom, PolygonGeom, and BPolygonGeom classes model a closed area defined by a series of line segments. The string format of polygon is "x1,y1 x2,y2,...".

PathGeom

The IPathGeom, PathGeom, and BPathGeom classes define a general path to draw or fill. The model and string format of a path is based on the SVG path element. The format is a list of operations. Each operation is denoted by a single letter. A capital letter implies absolute coordinates and a lowercase letter implies relative coordinates. Multiple operations of the same type may omit the letter after the first declaration.

- **Moveto**: move to the specified point. "M x,y" or "m x,y".
- **Lineto**: draw a line to the specified point: "L x,y" or "l x,y".
- **Horizontal Lineto**: draw a horizontal line at current y coordinate: "H x" or "h x".
- **Vertical Lineto**: draw a vertical line at the current x coordinate: "V y" or "v y".
- **Close**: draw a straight line to close the current path: "Z" or "z".
- **Curveto**: draws a Bezier curve from current point to x,y using x1,y1 as control point of beginning of curve and x2,y2 as control point of end of curve: "C x1,y1 x2,y2 x,y" or "c x1,y1 x2,y2 x,y".

Smooth Curveto: draws a Bezier curve from current point to x,y. The first control point is assumed to be the reflection of the second control point on the previous command relative to the current point. "S x2,y2 x,y" or "s x2,y2 x,y".

- **Quadratic Curveto**: draws a quadratic Bezier curve from current point to x,y using x1,y1 as control point: "Q x1,y1 x,y" or "q x1,y1 x,y".
- **Smooth Quadratic Curveto**: draws a quadratic Bezier curve from current point to x,y with control point a reflection of previous control point: "T x,y" or "t x,y".
- **Arc**: draws an elliptical arc from the current point to x,y: "A rx,ry x-axis-rotation large-arc-flag sweep-flag x y" or "a rx,ry x-axis-rotation large-arc-flag sweep-flag x y".

Refer to the W3 SVG spec (http://www.w3.org/TR/SVG/) for the formal specification and examples.

Transform

Transforms allow a new logical coordinate system to be derived from an existing coordinate system. The gx transform model is based on SVG and uses the exact string formatting. BTransform stores a list of transform operations:

- **Translate**: moves the origin by a tx and ty distance.
- **Scale**: scales the coordinates system by an sx and sy size.
- **Skew**: skew may be defined along the x or y axis.
- **Rotate**: rotate the coordinate system around the origin using a degree angle.

Image

The BImage class is used to manage bitmap images. Image's are typically loaded from a list of ords which identify a list of files (GIF, PNG, and JPEG supported). When more than file is specified, the image is composited using alpha transparency from bottom to top (useful for "badging" icons). Images may also be created in memory and "painted" using the `Graphics` API.

The framework will often load images asynchronsouly in a background thread to maintain performance. Developers using `BImages` directly can poll to see if the image is loaded via the `isLoaded()` method. Use the `sync()` method to block until the image is fully loaded. Animated GIFs require that the developer call `animate()` at a frame rate of 10frames/second. Typically developers should display images using BLabel which automatically handles async loading and animation.

The framework caches images based on their size and how recently they are used. You may use the Image Manager spy page to review the current cache.

Graphics

Painting to a device is encapsulated by the Graphics class. The primitive paint operations are:

- **fill(IGeom)**: filling a geometry involves painting a geometry's interior area with the current brush. Remember a brush can be a solid color, a gradient, or texture.
- **stroke(IGeom)**: stroking a geometry is to draw its outline or line segments. Stroking uses the current pen to derive the "stroke geometry" based on the pen's width and style. Then the interior of the stroke is filled using the current brush.
- **drawString()**: this draws a set of characters to the device. The shape of the characters is derived from the current font. The interior of the font is filled with the current brush. Note: we don't currently support stroking fonts like SVG.
- **drawImage()**: this draws an bitmap image to the device; the image may be scaled depending on the parameters and/or current transform.

All paint operations perform compositing and clipping. Compositing means that colors are combined as painting occurs bottom to top. For example drawing a GIF or PNG file with transparent pixels allows the pixels drawn underneath to show through. Alpha transparency performs color blending with the pixels underneath. Clipping is the processing of constraining paint operations to a specified geometry. Any pixels from a paint operation which would be drawn outside the clip geometry are ignored. Use the `clip()` method to clip to a specific region.

Often it is necessary to save the current state of the graphics to perform a temporary paint operation, and then to restore the graphics. An example is to set a clip region, paint something, then restore the original clip. The `push()` and `pop()` are used to accomplish this functionality by maintaining a stack of graphics state. You should always call `pop()` in a `try finally` block to ensure that your code cleans up properly.

Bajaui Widget Toolkit

Overview

The bajaui module contains a widget toolkit for building rich user interfaces. This toolkit is built using the Niagara component model. Widgets are `BComponents` derived from BWidget. Widgets define basic UI functions:

- **Layout**: defines the layout model - how widget trees are positioned on the graphics device
- **Painting**: defines how widgets paint themselves using graphical composition and clipping
- **Input**: defines how user widgets process user input in the form of mouse, keyboard, and focus events
- **Data Binding**: defines are how widgets are bound to data sources

Widgets are organized in a tree structure using the standard component slot model. Typically the root of the tree is a widget modeling a top level window such as BFrame or BDialog.

Layout

All widgets occupy a rectangular geometry called the bounds. Bounds includes a position and a size. Position is a x,y point relative to the widget parent's coordinate system. Size is the width and height of the widget. The widget itself defines its own logical coordinate system with its origin in the top left corner, which is then used to position its children widgets. Every widget may define a preferred size using `computePreferredSize()`. Layout is the process of assigning bounds to all the widgets in a widget tree. Every widget is given a chance to set the bounds of all its children in the doLayout() callback. When a layout refresh is needed, you may call `relayout()`. The relayout call is asynchronous - it merely enqueues the widget (and all its ancestors) for layout at some point in the near future.

Panes

Widget's which are designed to be containers for child widgets derive from the BPane class. A summary of commonly used panes:

- BCanvasPane: used for absolute positioning (discussed below);
- BBorderPane: is used to wrap one widget and provide margin, border, and padding similar to the CSS box model.
- BEdgePane: supports five potential children top, bottom, left, right, and center. The top and bottom widgets fill the pane horizontally and use their preferred height. The left and right widgets use their preferred width, and occupy the vertical space between the top and bottom. The center widget gets all the remainder space.
- BGridPane: lays out it children as a series of columns and rows. Extra space in the rows and columns is configurable a number of ways.
- BSplitPane: supports two children with a movable splitter between them.
- BTabbedPane: supports any number of children - only one is currently selected using a set of tabs.
- BScrollPane: supports a single child that may have a preferred size larger than the available bounds using a set of scroll bars.

Absolute Layout

Every widget also has a frozen property called layout of type BLayout. The `BLayout` class is used to store absolute layout. Widgets which wish to use absolute layout should be placed in a `BCanvasPane`. BLayout is a simple with the following string format "x,y,width,height". Each value may be a logical coordinate within the parent's coordinate space or it may be a percent of the parent's size. Additionally width and height may use the keyword "pref" to indicate use of preferred width or height. Examples include "10,5,100,20" "0,0,30%,100%", and "10%,10%,pref,pref".

Lastly the keyword "fill" may be used as a shortcut for "0,0,100%,100%" which means fill the parent pane. Fill is the default for the layout property which makes it easy to define layers and shapes.

Painting

All widgets are given a chance to paint themselves using the `paint(Graphics)` callback. The graphics is always translated so that the origin 0,0 is positioned at the top, left corner of the widget. The graphic's clip is set to the widget's size. Widget's with children, should route to paintChild() or `paintChildren()`. Painting follows the gx compositing rules. Alpha and transparent pixels blend with the pixels already drawn. Widgets are drawn in property order. So the first widget is drawn first (at the bottom), and the last widget drawn last (on the top). Note that hit testing occurs in reverse order (last is checked first). Effective z-order is reverse of

property order (consistent with SVG).

Input

User events are grouped into keyboard input, mouse input, and focus events. The following events are defined for each group:

BKeyEvent

- keyPressed
- keyReleased
- keyTyped

BMouseEvent

- mouseEntered
- mouseExited
- mousePressed
- mouseReleased
- mouseMoved
- mouseDragged
- mouseDragStarted
- mouseHover
- mousePulse
- mouseWheel

BFocusEvent

- focusGained
- focusLost

Design Patterns

Some complicated widgets have mini-frameworks all to their own. These include BTable, BTree, BTreeTable, and BTextEditor. All of these widgets use a consistent design pattern based on a set of support APIs:

- **Model**: defines the underlying logical model of the widget visualization
- **Controller**: processes all user input events, handles popup menus, and manages commands
- **Renderer**: responsible for painting the widget
- **Selection**: manages the current selection of the widget

Commands

The bajaui module provides a standard API for managing user commands using the Command and ToggleCommand classes. Commands are associated with one or more widgets which invoke the command. Typically this association happens by using a special constructor of the widget such as `BButton(Command cmd)` or using a `setCommand()` method. `Commands` are commonly used with BButton and BActionMenuItem. `ToggleCommands` are commonly used with BCheckBox, BRadioButton, BCheckBoxMenuItem, and BRadioButtonMenuItem.

Commands provide several functions. First they provide a centralized location to enable and disable the command. It is common for a command to be available via a menu item, a toolbar button, and a popup menu. By enabling and disabling the command all the widgets are automatically enabled and disabled.

Commands also provide a standard mechanism used for localization via the lexicon APIs. If one of the module or lexicon constructors is used the command automatically loads its visualization from a lexicon using a naming pattern: keyBase+".label", keyBase+".icon", keyBase+".accelerator", and keyBase+".description". The icon value should be an ord to a 16x16 png file. Widgets created with the Command will automatically set their properties accordingly.

The Command API also defines the basic framework for undo and redo. Whenever a command is invoked via the `invoke()` method, the Command can return an instance of CommandArtifact to add to the undo stack. Commands which don't support undo can just return `null`.

Data Binding

All widgets may be bound to zero or more data sources using the BBinding class. Bindings are added to the widget as dynamic child slots. You may use the `BWidget.getBindings()` to access the current bindings on a given widget. Bindings always reference a data source via an ord. The `BBinding` API defines the basic hooks given to bindings to *animate* their parent widget.

The most common type of binding is the BValueBinding class. Value binding provides typical functionality associated with building real-time graphics. It supports mouse over status and right click actions. Additionally it provides a mechanism to animate any property of its parent widget using `BConverters` to convert the target object into property values. Converters are added as dynamic properties using the name of the widget property to animate. For example to animate the "text" property of a `BLabel` you might add a `ObjectToString` converter to the binding using the property name "text".

Performance Tuning

The gx and bajaui toolkits are built using the AWT and Java2D. A key characteristic of performance is based on how widgets are double buffered and drawn to the screen. The following system properties may be used to tune widget renderering:

- **niagara.ui.volatileBackBuffer**: Defines whether the back buffer used for widget rendering uses `createVolatile()` or `createImage()`. Volatile back buffers can take advantage of Video RAM and hardware acceleration, non-volatile back buffers are located in system memory. Note: this feature is currently disabled.
- **sun.java2d.noddraw**: Used to disable Win32 DirectDraw. Often disabling DirectDraw can correct problems with flickering.

Workbench

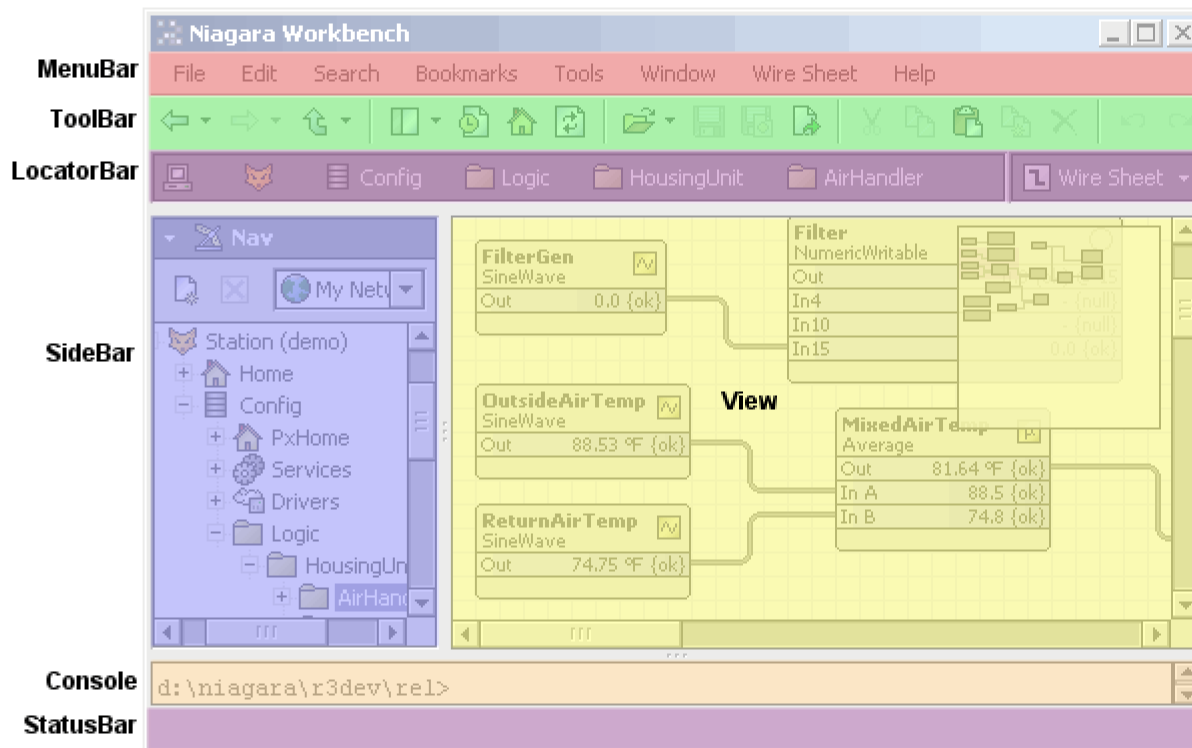
Overview

The workbench module define's the framework for building standardized user interfaces. The workbench provides enhancements to the bajaui widget toolkit:

- Standard layout with menu, toolbar, sidebars, and view;
- Standard browser based navigation model with an active ord, back, and forward;
- Bookmarking;
- Tabbed browsing;
- Options management;
- Plugin APIs;
- Ability to customize using WbProfiles;
- Enables both desktop and browser based applications;

Note: The term *workbench* applies both to the actual application itself as well as the underlying technology used to build customized applications. As you will see, all apps are really just different veneers of the same workbench customized using WbProfiles.

Layout



The illustration aboves shows the key components of the Workbench layout:

- **MenuBar**: The bar of standard and view specific pulldown menus;
- **ToolbarBar**: The bar of standard and view specific toolbar buttons;
- **LocatorBar**: Visualization and controls for active ord;
- **SideBar**: Pluggable auxiliary bars including navigation and palette;
- **View**: The main tool currently being used to view or edit the active object;
- **Console**: Used to run command line programs such as the Java compiler;
- **StatusBar**: Standard location to display status messages;

The `BWbShell` class is used to model the entire workbench window (or the applet in a browser environment). The `getActiveOrd()` method provides access to the current location, and `hyperlink()` is used to hyperlink to a new ord.

Browser Based Navigation

The fundamental navigation model of workbench is like a web browser. A web browser always has a current URL. As the current URL is changed, it fetches and displays the contents of that URL. A history of URLs is remembered allowing back and forward navigation. Most web browsers also allow the user to bookmark URLs for quick retrieval.

The workbench follows the same model. Instead of a URL to identity current location, the workbench uses an ord. The ord currently being viewed is called the *active ord*. Every ord resolves to a `BObject`. The target of the active ord is called the *active object*.

`BWbViews` are the plugins used to work with the active object. Views are the primary content of the workbench and provide a user interface to view or edit the active object. The workbench discovers the available views by searching the registry for `WbViews` registered on the active object.

The workbench provides a ready to use bookmark system that allows users to save and organize ords as bookmarks. Bookmarks are also used to store NavSideBar and FileDialog favorites. The bookmark system provides a public API via the javax.baja.ui.bookmark package.

Workbench also provides tab based browsing similar to modern browsers such as Mozilla and FireFox. Most places in the interface which allow double clicking to hyperlink support Ctrl+double click to open the object in a new tab. Also see the File menu for the menu items and shortcuts used to manupilate and navigate open tabs.

WbPlugins

The workbench is fundamentally a command and navigation shell, with all of it's functionality provided by plugins. The plugins available to the workbench are discovered by searching the registry for the appriopate type (WbProfiles allow further customization). This means that installing a new workbench plugin is as simple as dropping in a module.

All plugins subclass `BWbPlugin`, which is itself a `BWidget`. The following common plugins are discussed in the following sections:

- WbViews
- WbFieldEditor
- WbSideBars
- WbTools

WbView

Views are the workhorses of the workbench. Views provide the content viewers and editors for working with the active objects. Views also have the unique ability to do menu and toolbar merging. To implementing a new view plugin follow these rules:

- Create a subclass of `BWbView`, or if your view is on a `BComponent`, then subclass `BWbComponentView`;
- Override `doLoadValue(BObject, Context)` to update the user interface with the active object.
- If your view allows editing of the object, then call `setModified()` when the user makes a change which requires a save.
- If your view allows editing and `setModified()` has been called, override `doSaveValue(BObject, Context)` to save the changes from the user interface back to the active object.
- Register your view as an agent on the appriopate object type (or types):

```
<type name="AlarmConsole" class="com.tridium.alarm.ui.BAlarmConsole">
  <agent requiredPermissions="r"><on type="alarm:ConsoleRecipient"/></agent>
</type>
```

Writing a view for a `BIFile` typically involved reading the file's content for display on `doLoadValue()`, and writing back the contents on `doSaveValue()`.

Writing a `BWbComponentView` for a `BComponent` typically involves subscribing to the necessary component or components on `doLoadValue()`, and saving back changes on `doSaveValue()`. The `WbComponentView` class provides a series of `registerX()` methods for managing the view's subscriptions. Remember that if working with remote components, batching resolves, subscribes, and transactions can make significant performance improvements. Refer to Remote Programming for more information.

WbFieldEditor

`BSimple` `BStruct`

Field editors are similar to views, except they typically are smaller editors used to edit a or . Unlike views, a field editor never fills the view content area, but rather is used inside views like the PropertySheet.

The rules for building a field editor are very similar to views:

- Create a subclass of `BWbFieldEditor`.
- Override `doLoadValue(BObject, Context)` to update UI from object.
- Fire `setModified()` when the user makes a change.
- Override `doSaveValue(BObject, Context)` to update the object from the UI.
- Register your view as an agent on the appriopate object type (or type).

`BWbFieldEditor` also provides some convenience methods for displaying a dialog to input a specific `BObject` type. For example to prompt the user input a street address:

```
BStreetAddress addr = new BStreetAddress();
addr = (BStreetAddress)BWbFieldEditor.dialog(null, "Enter Address", addr);
if (addr != null) { /* do something! */ }
```

WbSideBar

Sidebars are auxiliary tools designed to be used in conjunction with the active view. Sidebars are displayed along the left edge of the view. Multiple sidebars can be open at one time. Unlike views, sidebars are independent of the active ord.

The rules for building a sidebar:

- Create a subclass of `BWbSideBar`.
- Provide display name and icon in your lexicon according to `TypeInfo` rules:

```
BookmarkSideBar.displayName=Bookmarks
BookmarkSideBar.icon=module://icons/x16/bookmark.png
```

WbTool

Tools are plugins to the workbench Tools menu. Tools provide functionality independent of the active ord. Typically tools are dialogs or wizards used to accompish a task. There are three types of tools:

- `BWbTool`: is the base class of all tools. It provides a single `invoke(BWbShell shell)` callback when the tool is selected from the Tools menu. Often invoke is used to launch a dialog or wizard.
- `BWbNavNodeTool`: is a tool which gets mounted into the ord namespace as "tool:{typespec}|slot:/". Selecting the tool from the Tools menu hyperlinks as the tool's ord and then standard `WbViews` are used to interact with the tool. Typically in this scenerio the tool itself is just a dummy component used to register one or more views.
- `BWbService`: is the most sophisticated type of tool. Services are `WbNavNodeTools`, so selecting them hyperlinks to the tool's ord. Services also provide the ability to run continuously in the background independent of the active ord. This is useful for monitoring tools or to run drivers locally inside the workbench VM. Services can be configured to start, stop, and auto start via the WbServiceManager.

The rules for building a tool:

- Create a subclass of `BWbTool`, `BWbNavNodeTool`, or `BWbService`.
- Provide display name and icon in your lexicon according to `TypeInfo` rules:

```
NewModuleTool.displayName=New Module
NewModuleTool.icon=module://icons/x16/newModule.png
```

WbProfiles

The `BWbProfile` class provides the ability to create new customized versions of the workbench. WbProfile provides hooks to replace

all of the standard layout components such as the MenuBar, ToolBar, LocatorBar, and StatusBar. Plus it provides the ability to customize which views, sidebars, and tools are available. Using WbProfiles you can quickly create custom applications that provide just the functionality needed for your domain.

You can launch workbench with a specific profile via a command parameter: `wb -profile:{typespec}`.

Note that if you wish to create an application that runs in the web browser you will need to subclass `BWbWebProfile`.

Web

Overview

The web module is used to provide HTTP connectivity to a station via the `BWebService`. The web module provides a layered set of abstractions for serving HTTP requests and building a web interface:

- **Servlet**: a standard `javax.servlet` API provides the lowest level of web integration.
- **ServletView**: is used to provide object views in a web interface.
- **Web Workbench**: is technology which enables the standard workbench to be run in a browser.
- **Hx**: is technology used to build web interfaces using only standards: HTML, JavaScript, and CSS.

Servlet

Niagara provides a standard javax.servlet API to service HTTP requests. The `WebOp` class is used to wrap `HttpServletRequest` and `HttpServletResponse`. `WebOp` implements `Context` to provide additional Niagara specific information. These APIs form the basis for the rest of the web framework.

The `BWebServlet` component is used to install a basic servlet which may be installed into a station database. The `servletName` is used to define how the servlet is registered into the URI namespace. A `servletName` of "foo" would receive all requests to the host that started with "/foo". Servlets are automatically registered into the URI namespace on their component `started()` method and unregistered on `stopped()`. The `service()` or `doGet()/doPost()` methods are used to process HTTP requests.

Note: The current `javax.servlet` implementation is based on version 2.4. The following interfaces and methods are not supported:

- **javax.servlet.Filter**: Unsupported interface
- **javax.servlet.FilterChain**: Unsupported interface
- **javax.servlet.FilterConfig**: Unsupported interface
- **javax.servlet.RequestDispatcher**: Unsupported interface
- **javax.servlet.ServletContext**: Unsupported methods
  - getNamedDispatcher(String)
  - getRequestDispatcher(String)
  - getResource(String)
  - getResourceAsStream(String)
  - getResourcePaths(String)
  - getServlet(String)
  - getServlets()
  - getServletContextName(String)
- **javax.servlet.ServletRequest**: Unsupported method
  - getRequestDispatcher(String)
- **javax.servlet.ServletRequestAttributeListener**: Unsupported interface
- **javax.servlet.ServletRequestListener**: Unsupported interface
- **javax.servlet.http.HttpSessionActivationListener**: Unsupported interface
- **javax.servlet.http.HttpSessionAttributeListener**: Unsupported interface
- **javax.servlet.http.HttpSessionListener**: Unsupported interface

ServletView

The web framework follows an object oriented model similar to the workbench. The user navigates to objects within the station using ords. One or more web enabled views are used to view and edit each object.

When navigating objects using ords, Niagara must map ords into the URI namespace. This is done with the URI format of "/ord?*ord*".

The `BServletView` class is used to build servlets that plug into the ord space using the "view:" scheme. For example if you wish to display an HTML table for every instance of component type "Foo", you could create a ServletView called "FooTable". Given an

instance of "Foo", the URI to access that view might be "/ord?slot:/foo3|view:acme:FooTable". The `WebOp` passed to `BServletView.service()` contains the target object being viewed (note `WebOp` subclasses from `OrdTarget`).

Web Workbench

A nice feature of Niagara's web framework is the ability to run the entire workbench right in a browser. This *web workbench* technology allows almost any view (or plugin) to run transparently in both a desktop and browser environment. The following process illustrates how web workbench works:

1. User requests a workbench view for a specific object via its ord.
2. An HTML page is returned that fills the entire page with a small signed applet called wbapplet.
3. The wbapplet is hosted by the Java Plugin which must be preinstalled on the client's machine.
4. The wbapplet loads modules from the station as needed, and caches them on the browser's local drive.
5. The workbench opens a fox connection under the covers for workbench to station communication.
6. The workbench displays itself inside the wbapplet using the respective WbProfile and WbView.

Web workbench technology allows a sophisticated UI to be downloaded to a user's browser straight out of a Jace. It is downloaded the first time and cached - subsequent access requires only the download of wbapplet (13kb). Development for web versus desktop workbench is completely transparent. The only difference is that the `BWbProfile` used for a web interface must subclass from `BWbWebProfile`. Some functionality is limited only to the desktop like the ability to access the console and Jikes compiler. Also note that web workbench session is limited to a specific station. So it doesn't make sense to navigate to ords outside that station such a "local:|file:".

Note: in order for web workbench to be used, the client browser machine must have access to the station's fox port. This may require the fox port to be opened up in the firewall.

Hx

There are cases where using the workbench is overkill or we don't wish to require the Java Plugin. For these use cases, Niagara provides the *hx* technology. Hx is a mini-framework used to build real-time web interfaces only with standard HTML, JavaScript, and CSS. See the hx chapter for details.

WebProfileConfig

The web experience of a given user is controlled via the `BWebProfileConfig` class. `WebProfileConfig` is a MixIn added to every `User` component. The web profile determines whether web workbench or hx is used by specifying an WbProfile or HxProfile for the user.

Px

Overview

Px is a technology used to package a UI presentation as an XML file. A px file defines a tree of bajaui widgets and their data bindings. Any `BWidget` and `BBinding` may be used, including those custom developed by you. Typically px files are created using a WYSIWYG tool called the PxEditor, although they can also be handcoded or auto-generated.

Px files are always given a ".px" extension, and modeled with the `file:PxFile` type.

Px Views

A px file may be used in a UI via a variety of mechanisms:

- Navigating directly to a px file will display its presentation
- The px file may be attached to a component as a dynamic view
- Many px files can be automatically translated into HTML using hx

The `WbPxView` is the standard presentation engine for px files. `WbPxView` is the default view of `file:PxFile`, so you can use px files just like an HTML file - by navigating to one.

The `BPxView` class may be used to define *dynamic views* on components. Dynamic views are like dynamic slots, in that they are registered on an instance versus a type. A dynamic view is automatically available for every `BPxView` found in a component. Each `BPxView` provides an ord to the px file to use for that view. PxViews may be added through the workbench or programmatically. Since the bindings within a px file are always resolved relative to the current ord, you can reuse the same px file across multiple components by specifying bindings with relative ords.

If all the widgets used in a px file have a translation to hx, then the entire px file can be automatically translated into HTML for hx users. See hx for more details.

PxMedia

As a general rule any `BWidget` is automatically supported when viewing a px file. However, viewing a px file in hx only supports a subset of widgets (those that have a hx agent). This means that you must target the lowest common denominator when creating px presentations. The target media for a px presentation is captured via the `BPxMedia` class. Both the px file and the PxView can store a PxMedia type. Currently there are only two available media types: `workbench:WbPxMedia` and `hx:HxPxMedia`. The PxEditor will warn you if you attempt to use widgets and bindings not supported by the target media.

API

The bajaui module provides a standard API for serializing and deserializing a widget tree to and from it's XML representation. The `PxEncoder` class writes a widget tree to an XML document. `PxDecoder` is used to read an XML document into memory as a widget tree.

Syntax

The bog XML format is optimized to be very concise with equal weight given to both read and write speed. The px XML format is designed to be more human usable. All px files have a root `px` element with a required `version` and optional `media` attribute. Within the root `px` element is an `import` element and a `content` element.

The `import` section contains a list of `module` elements. Each `module` specifies a Niagara module name using the `name` attribute. This module list is used to resolve type names declared in the `content` section.

The `content` section contains a single element which is the root of the px file's widget tree. Each component in the tree uses a type name as the element name. These type names are resolved to fully specified type specs via the import section.

Frozen simple properties of each component are declared as attributes in the component's start tag. Complex and dynamic slots are specified as children elements. The name of the slot inside the parent component may be specified using the `name` attribute. Dynamic simple properties specify their string encoding using the `value` attribute.

Example

The following example shows a BoundLabel placed at 20,20 on a CanvasPane, which is itself nested in a ScrollPane. Note since the CanvasPane is the value of ScrollPane's frozen property `content`, it uses the `name` attribute. Note how frozen simple properties like `viewSize`, `layout`, and `ord` are defined as attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<px version="1.0" media="workbench:WbPxMedia">
<import>
  <module name="baja"/>
  <module name="bajaui"/>
  <module name="converters"/>
  <module name="gx"/>
  <module name="kitPx"/>
</import>
<content>
<ScrollPane>
  <CanvasPane name="content" viewSize="500.0,400.0">
    <BoundLabel layout="20,20,100,20">
      <BoundLabelBinding ord="station:|slot:/Playground/SineWave" statusEffect="none">
        <ObjectToString name="text"/>
      </BoundLabelBinding>
    </BoundLabel>
  </CanvasPane>
</ScrollPane>
</content>
</px>
```

Hx

Overview

The hx module defines the framework for building HTML-based user interfaces using HTML, CSS, JavaScript, and XmlHttp.

Hx is designed to approximate the same paradigms that exist for developing user interfaces in the Workbench enviornment, such as Views, FieldEditors, and Profiles. It's main goal is try and transparently produce lightweight HTML-only interfaces automatically based on the workbench views. Limited support exists for standard views like the Property Sheet, but Px is the main reuse target.

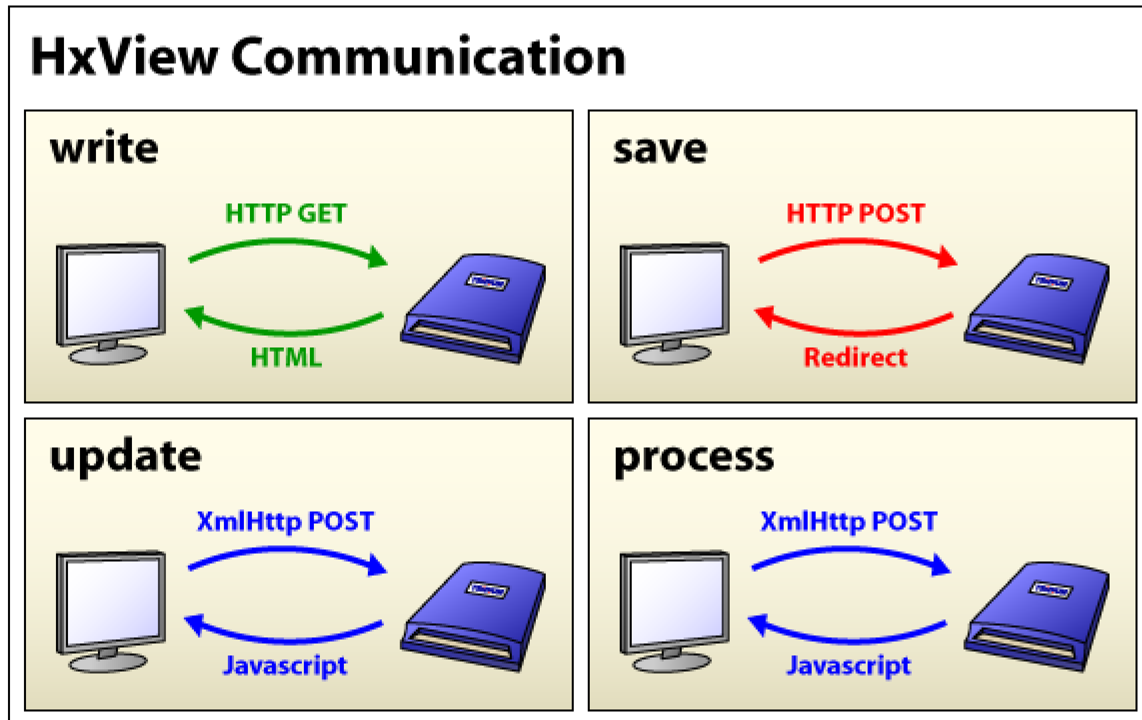
If you are not familiar with how interfaces are designed in workbench you should read the Workbench documentation before continuing.

- HxView Details of HxView
- HxOp Details of HxOp
- HxProfile Details of HxProfile
- Events Detail of Events
- Dialogs Details of Dialogs
- Theming Details of Theming

Hx - HxView

HxView Overview

HxView provides the content viewers and editors for working with the active objects. As you have probably guessed, HxView is the Hx equivalent of WbView. HxView is designed to produce and interact with a snippet of HTML. BHxProfile takes one or more HxViews, adds some supporting markup plus some chrome, and produces a complete HTML page.



From the diagram, a HxView:

- Must have logic to render a HTML snippet from an object (write). This is synonymous to BWbView.doLoadValue().
- May have logic to save changes back to the object (save). This is synonymous to BWbView.doSaveValue().
- May have logic to periodically update the HTML snippet (update).
- May have logic to respond to client background requests (process).

The name in parenthesis at the end of each bullet is the corresponding method in HxView responsible for that behavior. Details on each method can be found below. The HxProfile is responsible for building the containing HTML around each HxView.

Example

The details of each method can be found at the end of this document, but lets take a simple example to walk through the API:

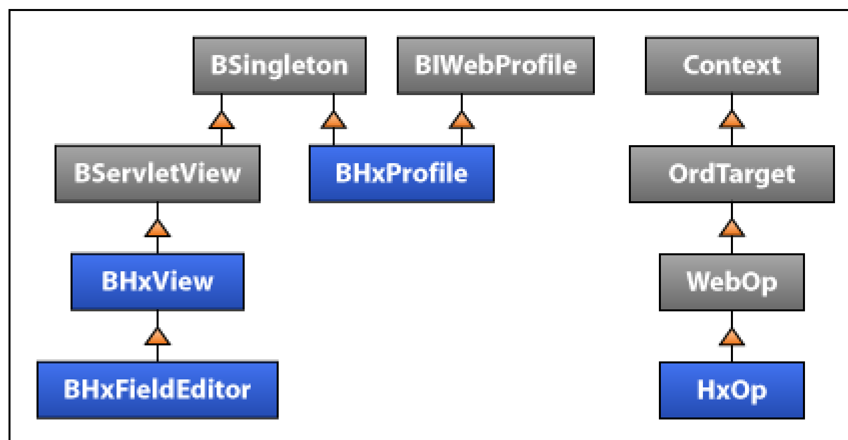
```
public class BFooView extends BHxView
{
  public static final BFooView INSTANCE = new BFooView();

  public Type getType() { return TYPE; }
  public static final Type TYPE = Sys.loadType(BFooView.class);

  protected BFooView() {}

  public void write(HxOp op) throws Exception
```

```
    {
      BFoo foo = (BFoo)op.get();
      HtmlWriter out = op.getHtmlWriter();
      out.w("Current name: ").w(foo.getName()).w("<br/>");
      out.w("<input type='text' name='").w(op.scope("name")).w("'");
      out.w(" value='").w(foo.getName()).w("' /><br/>");
      out.w("<input type='submit' value='Submit' />");
    }

    public BObject save(HxOp op) throws Exception
    {
      BFoo foo = (BFoo)op.get();
      foo.setName(op.getFormValue("name"));
      return foo;
    }
  }

  // Register this view on BFoo in module-include.xml
  <type name="FooView" class="bar.BFooView">
    <agent><on type="bar:Foo"/></agent>
  </type>
```

Assuming the current name in our `BFoo` object is "Peter Gibbons", the above code will produce the following HTML (ignoring the profile):

```
Current name: Peter Gibbons<br/>
<input type='text' name='name' value='Peter Gibbons' /><br/>
<input type='submit' value='Submit' />
```

If you are familiar with Niagara AX and HTML, this should be pretty straightforward. Let's walk through it though. Here's the class heirarchy of the main hx components:



The first thing to note is that `BHxView` extends `BServletView`, which extends `BSingleton`, which requires a `public static final INSTANCE` variable for all concrete implementations. If you have ever programmed Servlets before, you'll know that a Servlet must be re-entrant, and `HxViews` follow the same model. The `INSTANCE` object is what will be used to handle requests. (The `protected` constructor is a convention used to enforce the singleton design pattern). Since `HxView` can't maintain state while its doing its thing, we need to stash stuff somewhere - thats where HxOp comes in. We won't get into the details of `HxOp` yet, just be aware that anything I need to know about my current request or response is very likely accessible via the op.

Producing the HTML

Let's move on to the lifecycle of an `HxView`. The first thing a view will do is render its HTML to the client. This occurs in the `write()` method. By default Hx documents use the XHTML 1.0 Strict DOCTYPE. So you are encouraged to use valid XHTML in your `write` method. Since `HxViews` are designed to be chromable, and compositable, you'll also only write the markup that directly pertains to this view in your `write` method. Here are the things you should take note of about `write`:

- Think `BWbView.doLoadValue()`
- Only write the HTML that directly pertains to this view.
- You should always use `op.scope()` when writing a form element name. We'll get to why you should do that in 'Writing Reusable HxViews' down below.
- This is just a plain old HTML form, so all the normal form elements are applicable. And of course any other HTML.
- Just like a plain HTML file, `<input type='submit' />` is used to submit changes back to the server. The Hx framework will take care of building the `form` tag so this request gets routed to your `save()` method.

Saving Changes

Ok, my name is not "Peter Gibbons", so we need to be able to save something else back to the station. This is just as easy as writing my HTML, you simply implement a `save` method on your view. The request will automatically be routed, and all form data will be available from the `HxOp.getFormValue()` method.

So now if I view our example view in my browser, enter "Michael Bolton" and hit "Submit", the page will refresh to display:

```
Current name: Michael Bolton<br/>
<input type='text' name='name' value='Michael Bolton' /><br/>
<input type='submit' value='Submit' />
```

Technically, what happens, is the POST request gets routed to `save`, then Hx responds with a redirect back the same location. This forces the page contents to be requested on a GET request, avoiding double-posting problems.

Writing Reusable HxViews

Hx is designed to allow for reusing and nesting `HxViews` within other `HxViews`. To accomplish this, we need some type of scoping mechanism to avoid naming conflicts. Luckily, Hx handles this quite cleanly. Each HxOp that gets created has a name (explicit or auto-generated), which when combined with its parent tree, creates a unqiue path to each "instance" of a `HxView`. So if you take the this code:

```
public void write(HxOp op) throws Exception
{
  HtmlWriter out = op.getHtmlWriter();
  out.w("<input type='text' name='").w(op.scope("foo")).w("'/>");
  ...
}

public BObject save(HxOp op) throws Exception
{
  String foo = op.getFormValue("foo");
  ...
}
```

The resulting HTML could look something like this:

```
<input type='text' name='uid1.editor.uid5.foo'/>
```

`HxOp.getFormValue()` will automatically handle the "unscoping" for you. This allows any `HxView` to be nested anywhere without knowing its context. However, this only works if you follow a few rules:

- Always give sub-views a sub-op using `HxOp.make()` - there should always be a 1:1 ratio between `HxOps` and `HxViews`. See "Managing Subviews" below.

`HxOp.scope()` `name`

Always use when writing the attribute for a form control.

- When using the auto name constructor of `HxOp`, names are created by appending the current counter value to a string ("uid0", "uid1", etc). So its very important that the order in which `HxOps` are created is always the same in write/save/update/process so the produced paths will always the same. Otherwise views will not be able to correctly resolve their control values.

Managing Subviews

Hx does not contain any notion of containment, so composite views are responsible for routing all write/save/update/process requests to its children:

```
public class BCompositeView extends BHxView
{
  public void write(HxOp op) throws Exception
  {
    BSubView.INSTANCE.write(makeSubOp(op));
    ...
  }

  public BObject save(HxOp op) throws Exception
  {
    BFoo foo = BSubView.INSTANCE.write(makeSubOp(op));
    ...
  }

  public void update(HxOp op) throws Exception
  {
    BSubView.INSTANCE.update(makeSubOp(op));
  }

  public boolean process(HxOp op) throws Exception
  {
    if (BSubView.INSTANCE.process(makeSubOp(op)))
      return true;
    return false;
  }

  private HxOp makeSubOp(HxOp op)
  {
    BFoo foo;
    ...
    return op.make(new OrdTarget(op, foo));
  }
}
```

Don't forget to always create a sub-op to your child views so the Hx framework can strut its stuff.

Writing HxView Agents for WbViews

Another feature of the hx framework is a transparent transition from the Workbench environment to the hx environment. For example, if you have created a `WbView` called `WbFooView`, all references to that view can be made to transparently map to your hx implementation. You just need to register your `HxView` directly on the `WbView`, and expect the input argument to be the same type as loaded into the `WbView`:

```
// Register WbView agents directly on the View
<type name="HxFooView" class="foo.BHxFooView">
```

```
  <agent><on type="foo:WbFooView"/></agent>
</type>

public void write(HxOp op)
  throws Exception
{
  // Assume object is what the corresponding WbView would
  // receive in its doLoadValue() method.
  BFoo foo = (BFoo)op.get();
  ...
}
```

Then this ord will work correctly in both enviornments:

```
station:|slot:/bar|view:foo:WbFooView
```

Also note that if your view is the default view on that object, the default ord will choose the correct view as well:

```
station:|slot:/bar
```

Writing HxView Agents for Px Widgets

Similar to creating `WbView` agents, a `BHxPxWidget` is responsible for creating an hx representation of a `BWidget` used in a Px document. Note that `BHxPxWidget` works differently from a typical `HxView` in that op.get() will return the `BWidget` that this agent is supposed to model. The widget will already be mounted in a `BComponentSpace` and any bindings will already be active and leased. Also note that in `module-include.xml` this type should be registered as agent on the BWidget it is supposed to model.

```
// Register PxWidget agents directly on the Widget
<type name="HxPxLabel" class="com.tridium.hx.px.BHxPxLabel">
  <agent><on type="bajaui:Label"/></agent>
</type>

public void write(HxOp op)
  throws Exception
{
  // OrdTarget is the widget we want to model
  BLabel label = (BLabel)op.get();

  HtmlWriter out = op.getHtmlWriter();
  out.w(label.getText());
}
```

Uploading Files with Multi-part Forms

Hx supports file uploading by using the multi-part encoding for form submission. This capability is only supported for standard form submissions. You may upload one or more files along side the rest of your form. The files are stored in a temporary location on the station, and if not moved, will be automatically deleted at the end of the request.

- Must call `op.setMultiPartForm()` to change form encoding.
- Uploaded files are accessible from `op.getFile()`, where the control name designates which file you want.
- If the file is not explicity moved to another location, it will be deleted at the end of the request.

Let's take an example:

```
public void write(HxOp op) throws Exception
{
```

```
    op.setMultiPartForm();
    out.w("<input type='file' name='someFile' />");
  }

  public BObject save(HxOp op) throws Exception
  {
    BIFile file = op.getFile("someFile");
    FilePath toDir = new FilePath("^test");
    BFileSystem.INSTANCE.move(file.getFilePath(), toDir, op);
    return op.get();
  }
```

This code will upload a file to a temporary file, accessible as "someFile", and move it to another location so that it will not be deleted at the end of the request.

HxView Methods in Detail

write

Write is used to output the HTML markup for the current view. `HxViews` should only write the markup that directly pertains to itself. Avoid writing any containing markup - this is handled by the parent view or the profile. Especially avoid writing outer tags like `html`, `head`, and `body` - these are handled by the profile.

There is only one `form` tag in an hx page, and is written by the profile. `HxViews` should never write their own `form` blocks. So by design, the entire page content is encoded for `save` and `Events`. Naming collisions are handled automatically using the `HxOp` scoping rules (see 'Writing Reusable HxViews' above for more info on scoping).

The `write` method is always called on an HTTP GET request. However, if its written correctly (which typically means escaping quotes properly), it may be reused by `update` or `process` if it makes sense.

save

Save is used to save changes made from the view back to the target object. This is usually just a standard response to a form post, where the form values are accessed using `HxOp.getFormValue()`. Views on `BSimples` should return a new instance based on their new value. Views on `BComponents` should modify the existing instance and return that instance.

After a save is handled, a redirect is sent back to the browser to the current location. This is used to refresh the current page values, but more importantly to avoid double-posting problems. Content is always be requested on a GET request (and handled by `write`). You may choose to redirect back to a different URL using the `HxOp.setRedirect()` method.

The `save` method is always called on a standard HTTP POST form submit request. Both standard url-encoded and multi-part forms are supported. See 'Uploading Files with Multi-part Forms' above for info on multi-part forms.

update

Update is automatically called periodically on all views if at least one view was marked as dynamic (via `HxOp`). This is a background request made using JavaScript and XmlHttp. The content returned to the browser must be executable JavaScript. For example:

```
  public void write(HxOp op) throws Exception
  {
    op.setDynamic();
    HtmlWriter out = op.getHtmlWriter();
    out.w("<div id='time'>Current Time</div>");
  }

  public void update(HxOp op) throws Exception
  {
    HtmlWriter out = op.getHtmlWriter();
    out.w("var elem = document.getElementById('time');");
    out.w("elem.innerHTML = '").w(BAbsTime.now()).w("';");
```

```
}
```

Here, after the page is initially written, the browser will poll the station every five seconds running `update` on all the views. So this code will simply update the current time each time the station is polled.

process

Process is used to handle non-update background requests. A process request is targeted and serviced by a single `HxView`. The default implementation for process handles routing events to the correct view. See Events.

Note: If you override process, you must call super or event routing will fail.

Hx - HxOp

HxOp

HxOp maintains all the state for the current request, and provides the interface for creating and consuming a document. The original HxOp wraps the WebOp for the current request. Sub-views should be given a new HxOp from the current op via the HxOp.make() method. See 'Writing Reusable HxViews' in HxView.

Note: There should always be a one-to-one mapping of HxOps to HxViews.

- WebOp API
- HxOp API
- Servlet API

Hx - HxProfile

HxProfiles

The BHxProfile is used to customize the HTML page around the current HxView. The profile is responsible for writing out the outer HTML tags (html, head, and body), any custom markup, and the current view. It is important that your profile respect the order HxOps are created in these methods: writeDocument, updateDocument, processDocument, and saveDocument. If any HxView uses the auto name constructor of HxOp to create a unique path name, it must be called in the exact same order in order to resolve correctly.

HxProfile exposes customization hooks through convenience methods, so there is no need to handle the boilerplate code:

```
public class BMyProfile
  extends BHxProfile
{
  public static final BMyProfile INSTANCE = new BMyProfile();

  public Type getType() { return TYPE; }
  public static final Type TYPE = Sys.loadType(BMyProfile.class);

  protected BMyProfile() {}

  public void doBody(BHxView view, HxOp op)
    throws Exception
  {
    BHxPathBar.INSTANCE.write(op.make(op));
    displayError(op);
    view.write(op);
  }

  public void updateDocument(BHxView view, HxOp op)
    throws Exception
  {
    BHxPathBar.INSTANCE.update(op.make(op));
    view.update(op);
  }

  public boolean processDocument(BHxView view, HxOp op)
    throws Exception
  {
    if (BHxPathBar.INSTANCE.process(op.make(op)))
      return true;
    return view.process(op);
  }

  public void saveDocument(BHxView view, HxOp op)
    throws Exception
  {
    BHxPathBar.INSTANCE.save(op.make(op));
    view.save(op);
  }
}
```

Hx - Events

Events

Hx uses `Events` to provide background interaction between the server and the browser. Events always originate from the client browser, and must return executable javascript as the response (you are not required to return content). The html form is encoded and sent for every event fire, so `op.getFormValue()` can be used to query the browser page state.

`Events` are implemented on top of the `HxView.process` method, and therefore use the `XmlHttp` support implemented in the major browsers.

Command extends `Event` to add some convenience methods and a display name property. By convention `Commands` are triggered by the user (maybe by clicking on a button), while `Events` are triggered programmatically. Though in reality they are interchangeable.

Note: `javax.baja.hx.Command` is not the same class as the `javax.baja.ui.Command`.

Hx - Dialogs

Dialogs

Support for modal dialog boxes is provided with `Dialog` and is typically used from an `Command`:

```
class EditCommand extends Command
{
  public EditCommand(BHxView view)
  {
    super(view);
    dlg = new EditDialog(this);
  }

  public void handle(HxOp op) throws Exception
  {
    if (!dlg.isSubmit(op)) dlg.open(op);
    else
    {
      String name = op.getFormValue("name");
      String age  = op.getFormValue("age");

      BDude dude = (BDude)op.get();
      dude.setName(name);
      dude.setAge(Integer.parseInt(age));

      refresh(op);
    }
  }

  private EditDialog dlg;
}

class EditDialog extends Dialog
{
  public EditDialog(Command handler) { super("Edit", handler); }
  protected void writeContent(HxOp op) throws Exception
  {
    BDude dude = (BDude)op.get();
    HtmlWriter out = op.getHtmlWriter();

    out.w("<table>");
    out.w("<tr>");
    out.w(" <td>Name</td>");
    out.w(" <td><input type='text' name='").w(op.scope("name"));
    out.w("' value='").w(dude.getName()).w("'/></td>");
    out.w("</tr>");
    out.w("<tr>");
    out.w(" <td>Age</td>");
    out.w(" <td><input type='text' name='").w(op.scope("age"));
    out.w("' value='").w(dude.getAge()).w("'/></td>");
    out.w("</tr>");
```

```
      out.w("</table>");
    }
  }
```

Hx - Theming

Theming

All styling in hx is handled with CSS. The core colors and fonts are defined in `module://hx/javax/baja/hx/default.css`. In order for your view to use the default theme, you should write your markup like this:

```
<div class="controlShadow-bg myCustomClass" style="...">
...
</div>
```

This order is important. The default class should always come first in the selector list, and before any style tag (though you should avoid using style directly in your view) - so that styles are overridden correctly.

Note: `HxProfiles` that override the theme should always place their custom stylesheet last to make sure it overrides any stylesheets loaded during the `write()` phase.

Control

Overview

The control module provides normalized components for representing control points. All control points subclass from the BControlPoint base class. Control points are typically used with the driver framework to read and write points in external devices.

There are four normalized categories of data matching the four `BStatusValue` types. Within each of the four categories is a readonly component and a writable component. These eight components are:

Type	Mode	Data
BBooleanPoint	RO	Models boolean data with BStatusBoolean
BBooleanWritable	RW / WO	Models boolean data with BStatusBoolean
BNumericPoint	RO	Models numeric or analog data with BStatusNumeric
BNumericWritable	RW / WO	Models numeric or analog data with BStatusNumeric
BEnumPoint	RO	Models discrete values within a fixed range with BStatusEnum
BEnumWritable	RW / WO	Models discrete values within a fixed range with BStatusEnum
BStringPoint	RO	Models unicode strings with BStatusString
BStringWritable	RW / WO	Models unicode strings with BStatusString

Design Patterns

All control points use `BStatusValues` to represent their inputs and output. All points have one output called "out". The readonly points contain no inputs. Typically they model a value being read from a device via the driver framework.

The writable points all contain 16 inputs and a fallback value. These 16 inputs are prioritized with 1 being the highest and 16 being the lowest. The value to write is calculated by finding the highest valid input (1, 2, 3, down to 16). An input is considered valid if none of the following status bits are set: disabled, fault, down, stale, or null. If all 16 levels are invalid, then the fallback value is used. Note that the fallback value itself can have the null bit set in which case the point outputs null. The active level is indicated in the output as a status facet.

Each of the writable points reserves level 1 and level 8 for user invoked overrides. Level 1 is an emergency override which when invoked remains in effect permanently until the emergencyAuto action is invoked. Level 8 overrides are for normal manual overrides. Manual overrides may be timed to expire after a period of time, or may be explicitly canceled via the auto action. Whenever level 1 or 8 is the active level then the overridden status bit is set in the output. If a timed override is in effect then the overrideExpiration property indicates when the override will expire.

Extensions

Extensions provide building blocks to extend and change the behavior of control points. Every extension must derive from BPointExtension. They are added as dynamic properties on a control point. Extensions can process and modify the value of a control point whenever it executes. For example, an alarm extension can monitor the value and set the alarm bit of the output's status if an alarm condition was detected. A list of extensions include:

- BDiscreteTotalizerExt
- BNumericTotalizerExt
- BProxyExt
- BAlarmSourceExt
- BIntervalHistoryExt
- BCovHistoryExt

Extensions are always invoked in the order they are declared in the slot list. They may be reordered using the standard reorder API and workbench commands.

When the execute method is invoked on a BControlPoint, the `pointChanged(ControlPoint pt)` method is in turn invoked on each extension.

Note that when using extensions with driver proxy points, only the value being read is processed by extensions.

History

Overview

Refer to the javax.baja.history API.

The History module manages the storage, collection, and archiving of data logs (historical data). A data log in Niagara is often referred to as a Baja history (or history for short) and is an implementation of BIHistory. Within Niagara, histories can be accessed locally or remotely (via Niagara's Fox communication). The History API provides the basis for creating, configuring, modifying, accessing, and deleting histories. The Driver History API provides the means for archiving histories (pulling/pushing histories from one station to another).

In order to provide support for a database of histories in a Niagara station, the History Service must be added (BHistoryService). It is responsible for creating the database and enables collection and storage of histories in the database. Once the History Service is in place, the basis for managing access to histories in the database is through the History Space (BHistorySpace). Whenever you wish to gain access to a history, it is handled by resolving through the `BHistorySpace`. BHistoryDatabase is a local implementation of `BHistorySpace`. It handles opening and closing history files as they are needed and also provides efficient access to these files.

Access

As mentioned, in order to access histories in the database, you must first gain access to the database itself. This is done by resolving the history ord scheme (as defined by BHistoryScheme). The unique history scheme name is "history". Refer to the Naming documentation for details on Niagara's naming system. For example, if you want to access a history named "TestLog" in a station's database (the station being named "demo"), your ord would contain the query, "history:/demo/TestLog". You will notice that histories are organized by their source station (device), or BHistoryDevice.

When a history is retrieved from the database, it is always an implementation of BIHistory. `BIHistory` provides access to the following:

- The history's identification. Histories are uniquely identified by a String identification composed of two parts, the source device name and the history name. This identification information is encapsulated in BHistoryId. For example, if you have a history named "TestLog" and it is located uder the local station named "demo", the history id would be the combination of device (station) name and history name: "demo/TestLog".
  Note: For convenience when importing/exporting histories between Niagara stations (refer to Driver History), you can use the shorthand character '^' to refer to the parent device name. For example, if you are exporting a local history generated by the local station, the shorthand representation for the previous example would be: "^TestLog".
- Summary information about the history. This information is encapsulated in BHistorySummary. It provides such things as the history ID, number of records in the history, the timestamp of the first record in the history, and the timestamp of the last record in the history.
- The type of records in the history. This is normally a concrete type of BHistoryRecord which will be described in more detail later.
- The configuration of the history. This is defined in BHistoryConfig which will be described in more detail later.
- The data in the history itself. It provides support for scanning the records in the history, performing a time based query for records, and appending or updating records within the history.

A history contains records which are keyed by timestamp. A record is an instance of BHistoryRecord which supplies the timestamp key (records can always be identified by timestamp) and implements the BIHistoryRecordSet interface (always a set of 1 for a single history record). A BTrendRecord is a special extension of a `BHistoryRecord` which adds two more tidbits of information to a history record: trend flags (BTrendFlags) and status (BStatus). Trend flags are used to provide extra context information about the record data, such as the starting record, out of order records, hidden records, modified records, or interpolated records. The status ("ok", "alarm", "fault", etc.) is associated with the collected data value. The standard Niagara data value types are supported via extensions of `BTrendRecord`: BBooleanTrendRecord, BEnumTrendRecord, BNumericTrendRecord, and BStringTrendRecord.

Note: When a `BIHistory` is scanned or queried for its data records, it most often returns a `Cursor` (HistoryCursor) or a BICollection. When iterating through this `Cursor` or `BICollection`, it is important to note that it returns the same instance of `BHistoryRecord` for each iteration. This is done for performance reasons. So, if you need to store the records for later use as you

iterate through them, be sure to make a copy of the instance (you can use the `newCopy()` method).

You can also query the database via a history ordQuery as defined in HistoryQuery. This allows you to find histories and filter the data returned.

Configuration and Collection

When a user is ready to start logging data in Niagara, the most common way accomplish this is by adding a concrete instance of a history extension (BHistoryExt) to a control point. This is just like adding any point extension (BPointExtension) to a control point, extending its behavior. `BHistoryExt` is an extension of `BPointExtension`, however it also implements the BIHistorySource interface which allows it to be the creator of a history. `BHistoryExt` is an abstract class which provides the following among other things:

- The configuration of the history to create. This information is contained in a BHistoryConfig instance. It contains the following:
  - The unique identifier for the history within the entire system (`BHistoryId`).
  - The original source of the history.
  - The timezone where the history was originally collected.
  - The type of records contained in the history (i.e. `BBooleanTrendRecords`, `BNumericTrendRecords`, etc.).
  - The schema (BHistorySchema) for the records which allows the history to be read even if the original record type class has changed or is not available.
  - The amount of data that can be stored in the history (BCapacity).
  - The behavior when an attempt is made to write records to the (limited capacity) history that is already full (BFullPolicy).
  - The mechanism for storage of the history records (BStorageType). This defaults to a file.
  - The amount of time between records in the history (BCollectionInterval).
- The time period when the history extension should be collecting history records (BActivePeriod). This is normally a BBasicActivePeriod which allows the user to specify the days of the week and time of day that history records should be recorded.
- A definition of the pattern for deriving the name of the history created by the history extension. This property is of type BFormat and it can be static text or a simple pattern that allows the actual history name to be derived from the context.

There are two main types of `BHistoryExts` supported in the History module. These are the typed instances of BCovHistoryExt and BIntervalHistoryExt. `BCovHistoryExt` provides support for collecting history records triggered on changes to the value of the parent control point while `BIntervalHistoryExt` provides support for collecting history records based on a user defined fixed interval.

Compatibility

It is important to remember that there are two types of changes that an end user can make to a history extension (or `BIHistorySource`) to cause its history to be split (recreated with a new name). If the record type changes (i.e. a switch from numeric records to String records), this is an incompatible change. Another incompatible change is if the interval of collection changes. In both of these cases, the generated history will be split; the old history will keep its name, and the new history will have the same root name, but with a postfix ("_cfg#") appended to the end of it. For example, if the history "TestLog" encounters an incompatible change, the old history will keep its records and the name "TestLog", while any new records will be placed in a new history named "TestLog_cfg0". If yet another incompatible change occurs after the first, the next split will have the new history named "TestLog_cfg1", and so on.

Archiving

Refer to the Driver History documentation.

History Exceptions

The History API defines a few standard history exceptions. These all extend from HistoryException which is a BajaRuntimeException.

- A ConfigurationMismatchException is thrown when the properties of a `BIHistory` do not match the properties for that

history that are stored in the actual database.

- A DatabaseClosedException is thrown when an operation is attempted on a history database that is not open.
- A DuplicateHistoryException is thrown when an attempt is made to create a history with an id that already exists.
- A HistoryClosedException is thrown when a history is closed at a time when it is expected to be open.
- A HistoryDeletedException is thrown when an attempt is made to access a history that has been deleted.
- A HistoryNameException is thrown when an attempt is made to create a history with an invalid name.
- A HistoryNotFoundException is thrown when a history cannot be found in the history database.
- An IllegalConfigChangeException is thrown when an attempt is made to reconfigure a history in an unsupported way.
- An InvalidHistoryIdException is thrown when an attempt is made to open a history without a valid history id.

Alarm

Introduction

The Alarm module provides core functionality for lifecycle managment of alarms within the Baja framework. Alarms are used to indicate that some value is not within an appropriate or expected range. Alarms may be routed from the system to a variety of external sources, be it email, a printer or a console application.

Object Model

All alarms in the Baja Framework are generated by objects implementing the BIAlarmSource interface. Those alarms (BAlarmRecord) are then routed to the BAlarmService. The service for storing and routing of alarms. Alarms are then routed to one or more recipients (BAlarmRecipient) via their BAlarmClass.

Alarm Sources

While BIAlarmSource is an interface, most alarm sources are instances of `javax.baja.control.alarm.BAlarmSourceExt`, the alarm point extension. The alarm extension determines when it's parent point is in an alarmable condition, and uses the `AlarmSupport` class to take care of routing and issuing alarms to the alarm service. The alarm source updates the alarm when the parent point goes back to its normal condition as well as notifies the point that an acknowledgement has been received.

Objects implementing BIAlarmSource that have a status (BStatus) should use the following rules when setting the status bits.

- Generate Offnormal alarm: set the BStatus.ALARM and BStatus.UNACKED_ALARM bits.
- Generate Fault alarm: set the BStatus.ALARM, BStatus.UNACKED_ALARM and BStatus.FAULT bits.
- AckAlarm methods is called: if the alarm is the last one generated clear the BStatus.UNACKED_ALARM bit.
- Generate Normal alarm: clear the BStatus.ALARM and BStatus.FAULT bits.

Note that BStatus.UNACKED_ALARM should only be set if the BAlarmClass.ackRequired bit is set for that transition in the AlarmSource's AlarmClass. This can easily be ontained if using the AlarmSupport class by calling BAlarmSupport.ackRequired(BSourceState state).

Alarm Service

The `BAlarmService` coordinates routing of alarms within the framework. It routes alarms from their source to the appropriate recipients, and alarm acknowledgements from the recipients back to the source. The alarm service routes individual alarms via their alarm class. All alarm classes available to the system are maintained as slots on `BAlarmService`. The BAlarmService also maintains the Alarm Database. It is acessed though the `getAlarmDb()` method.

Alarm Class

The alarm classes, as stated above, are maintained as slots on the alarm service and serve to route alarms with similar sets ot properties along common routes - they serve as channels for like data. `BAlarmClass` manages the persistence of the alarms as needed via the alarm database. The AlarmClass manages the priority of an alarm and also which alarm require acknowledgement. Each alarm class can be linked to one or more alarm recipients.

Alarm Recipients

Alarm recipients are linked to an alarm class (from the `alarm` topic on the alarm class to the `routeAlarm` action on `BAlarmRecipient`.) Recipients may be configured to receive alarms only at certain times of day, certain days of the week, and receiving alarms of only certain transitions (eg. toOffnormal, toFault, toNormal, toAlert).

Three subclasses of `BAlarmRecipient` are worth noting: `BConsoleRecipient`, `BStationRecipient` and `BEmailRecipient`.

BConsoleRecipient

This recipient manages the transfer of alarms between the alarm history and the alarm console, i.e. it gets open alarms from the alarm history for the console and updates the history when they are acknowledged.

BStationRecipient

This recipient manages the transfer of alarms between the alarm service and a remote Niagara station.

BEmailRecipient

The email recipient is part of the email package. It allows alarms to be sent to users via email.

Lifecycle

Each alarm is a single BAlarmRecord that changes throughout its lifecycle. An alarm has four general states that it may be in:

1. New Alarm
2. Acknowledged Alarm
3. Normal Alarm
4. Acknowledged Normal Alarm

All alarms start as New Alarms and end as Acknowledged Normal Alarms. They may be acknowledged then go back to normal or go back to normal then be acknowledged.

An Alert is an alarm that does not have a normal state and thus its lifecycle consists of New Alarm and Acknowledged Alarm.

Alarm Routing Overview

New Alarms

1. BIAlarmSource generates an offnormalAlarm (or faultAlarm).
2. It is sent to the BAlarmService
3. BAlarmService routes it to its BAlarmClass.
4. The BAlarmClass sets the alarm's priority, ackRequired bit, and optional data.
5. It is then routed to any number of BAlarmRecipients.

The normal alarm is sent along this same path.

Alarm Acks

1. When a BAlarmRecipient acknowledges an alarm, the acknowledgement is sent to the BAlarmService
2. The BAlarmService routes back to the BIAlarmSource (if an ack is required).
3. The Alarm Acknowledgement is then routed to AlarmRecipients along the same path as a New Alarm.

Usage

Setup

The most basic piece needed is a control point. Then add an alarm extension from the alarm module palette. There are several types of extensions depending upon the type of point selected. The AlarmExtension are disabled by default. You must enabled toOffnormal or toFault alamrs and configure and enable the alarm algorithms.

An Alarm Service is also required. Depending on your needs, it may require some of the following slots:

- Any desired `BAlarmClass`es should be added.
- A `BConsoleRecipient` should be added if an alarm console is required.

Link any of the slots as needed. The alarm recipients must be linked to an alarm class in order to receive alarms from that alarm class.

To generate an alarm, go to a point with an alarm extension and put it an alarm condition.

Console Recipient / Alarm Console

To view all of the outstanding alarms in the system, double click on the console recipient on the alarm service. The alarm console manages alarms on a per point basis. Each row in the alarm console is the most recent alarm from a point. To view all the current alarms from that point, double click the row. To acknowledge an alarm, select the desired alarm and hit the ack button. An alarm is cleared from the alarm console when the alarm is acknowledged AND the point is in its normal state. To view more information about an unacknowledged alarm, right click and select View Details.

Station Recipient

A `BStationRecipient` allows sending alarms to remote Niagara stations. A remote station is selected from the stations you have configured in your Niagara Network. This recipient require that the remote station be properly configured in the Niagara Network.

Line Printer Recipient

A `BLinePrinterRecipient` allows printing of alarms on a Line Printer. This recipient is only availible on Win32 Platforms. It supports both local and remote printers.

Schedule

Overview

A schedule is effective or it is not. When it becomes effective, it will do something like fire an event or change an output. When a schedule is not effective, it will have some default configurable behavior.

Most schedules will be a hierarchy of many schedules. Container schedules combine the effective state of their descendants to determine effectiveness. Atomic schedules use some internal criteria to determine effectiveness. An example of an atomic schedule is the month schedule. It can be configured to be effective in some months and not in others.

Creating New Schedule Types

BAbstractSchedule

All schedules subclass this.

Subclassing. To create a new schedule type, one simply needs to implement methods `isEffective(BAbsTime)` and `nextEvent(BAbsTime)`. See the API documentation for details.

New Properties. Properties on new schedule types should have the `user_defined_1` flag set. This is important for properties who when changed, should cause supervisor (master) schedules to update their subordinates (slaves).

Output. If the new schedule is going to be used in a control schedule, it will be necessary to assign an effective value to it. A control schedule finds output by searching child schedules, in order, for the first effective schedule with a dynamic property named "effectiveValue". The effectiveValue may be 10 levels deep, it will be found. Just remember the order of schedules in a composite is important.

BCompositeSchedule

Composite schedules shouldn't need to be subclassed. However, they will be used (frequently) in building new schedule hierarchies.

These schedules perform a simple function, they determine their effective state by combining the effective state of their children. A composite can either perform a union or an intersection of it's children. A union means only one child has to be effective for the parent composite to be effective. An intersection means all children have to be effective.

Using Existing Schedules

There are six preconfigured schedules. The weekly schedules look like control objects, the calendar schedule helps configure special events that will be used by multiple schedules. Lastly, the trigger schedule enables sophisticated scheduling of topics (events) which can be linked to actions on other components.

BBooleanSchedule, BEnumSchedule, BNumericSchedule and BStringSchedule

These are all BWeeklySchedules whose output matches their name. There is one input who if linked and not null, completely overrides the schedule.

Example: Adding a special event

```
BDailySchedule specialEvent = new BDailySchedule(
    new BDateSchedule(5,BMonth.may,-1),
    BTime.make(11,0,0),
    BTime.make(12,0,0), //first excluded time
    BStatusBoolean.make(true));
myBooleanSchedule.addSpecialEvent(specialEvent);
```

Example: Adding to the normal weekly schedule

```
BDaySchedule day = myBooleanSchedule.get(BWeekday.monday);
day.add(BTime.make(11,0,0),BTime.make(12,0,0),BStatusBoolean.make(true));
```

Example: Retrieving all schedules in a normal weekday

```
BDaySchedule day = myBooleanSchedule.get(BWeekday.monday);
BTimeSchedule[] schedules = day.getTimesInOrder();
```

Example: Modifying the day schedule of a special event

```
BDailySchedule specEvent = (BDailySchedule)
myWeeklySchedule.getSpecialEvents().get("cincoDiMayo");
BDaySchedule day = specEvent.getDay();
```

Example: Retrieving all special events

```
BDailySchedule[] specEvents = myWeeklySchedule.getSpecialEventsChildren();
```

Legal special event schedule types:

- BDateSchedule
- BDateRangeSchedule
- BWeekAndDaySchedule
- BCustomSchedule
- BScheduleReference

BCalendarSchedule

This schedule has a boolean output. However, it's most common use is for special events in the four weekly schedules discussed above. The weekly schedule can store a special reference to any calendar in the same station and assign their own output to it.

Example: Adding a date schedule event

```
myCalendarSchedule.add("cincoDiMayo",aDateSchedule);
```

Legal event schedule types:

- BDateSchedule
- BDateRangeSchedule
- BWeekAndDaySchedule
- BCustomSchedule

BTriggerSchedule

This schedule fires an event when a schedule becomes effective. There is also an event signifying that a normal event has been missed.

Example: Add a date schedule event

```
myTriggerSchedule.getDates().add("cincoDiMayo",aDateSchedule);
```

Example: Add a trigger time

```
myTriggerSchedule.getTimes().addTrigger(11,00);
```

Legal event schedule types:

- BDateSchedule
- BDateRangeSchedule
- BWeekAndDaySchedule
- BCustomSchedule

Report

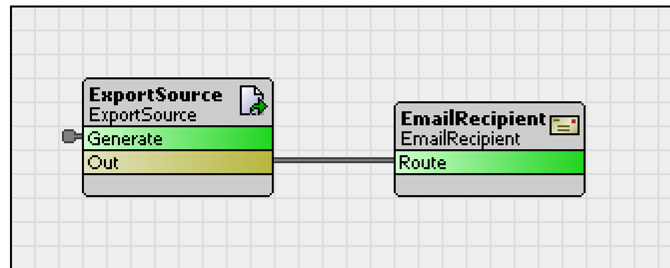
Introduction

The Report module provides facilities for running periodic background reports on a station.

ReportService

The ReportService provides a container for the components responsible for generating and routing reports. The process of generating a report is broken down into two components: BReportSource and BReportRecipient.

BReport Lifecycle



(ExportSource and EmailRecipient are concrete implementations for ReportSource and ReportRecipient, respectively.)

1. The `generate` action gets invoked on BReportSource. The action can be invoked manually or automatically via the built-in schedule property.
2. ReportSource creates a new BReport object which gets propagated to the ReportRecipient.
3. BReportRecipient handles routing the report to some destination.

BQL

Introduction

The Baja Query Language (BQL) is an SQL-like query language that provides a mechanism for identifying various sets of data. It provides an ad hoc way to search for data based on some criteria. By including BQL in an ord, the results can be easily book marked or embedded in graphics views. This makes BQL an excellent tool for building reports.

Select

The select query is the most common type of BQL query. It is very similar to the select statement in SQL. The syntax is as follows:

```
select <projection> from <extent> where <predicate> <having> <order by>
```

The select statement always returns a table even if the result is actually a single object.

Extent

The first concept to understand about the above query is the extent. The extent is specified in the "from" clause of the query. The extent works together with the ord base to determine the general set of objects in the result. The rest of the query only narrows the general result. This is best explained with a few examples.

```
slot:/a/b/c|bql:select name, toString from control:ControlPoint
```

In the above query, the base of the "bql" query is the "slot" query. The slot scheme is used to access data in a Baja component tree. In this case, "slot:/a/b/c" identifies the root of the tree where the BQL query processing will start. From that point, the query will recursively search the tree for components of type "control:ControlPoint". So, when the base of the "bql" query is a slot path, the path identifies the subtree that will be searched by the query, and the extent identifies the type of component to search for. This query would get the name and toString for all control points under the /a/b/c in the component tree.

```
history:|bql:select timestamp, value, status from /myStation/myHistory
```

In this query, the base of the "bql" query is a "history" query. The history scheme is used to access data in the Baja history database. In this case, "history:" identifies the entire set of histories in the database. The query extent "/myStation/myHistory" identifies a specific history in the database. This query would get the timestamp, value, and status of all records in the history with the id "/myStation/myHistory".

Projection

The projection is a comma separated list of the columns that will be returned in the result. Each element in the list must have a column specification and may have a display name specified with the 'as' clause. Beginning in Niagara 3.5, columns may be arbitrary expressions. The most frequent type of expression is a path expression, but you can also call scalar or aggregate functions.

```
select name, toString from baja:Component
select name as 'Point', out.value as 'Output Value', out.status from control:NumericPoint
select MAX(out.value), MIN(out.value) from control:NumericPoint
select (out.value * 100) + '%' as 'Percent' from control:NumericPoint
```

In the second query, we know that all numeric points have an "out" property that is a StatusValue. A StatusValue is a structure that contains both a status and a value. In this query, we use a path to dive into the structure and extract the value and status individually.

In the third query, we use two aggregate functions, MAX and MIN, to find the largest and smallest value of all the control:ControlPoints in our query. The result will only have one row. See the section on BQL functions for more details

In the fourth query, we perform a calculation on the out.value to make it a percent, and then append the '%' character to the result so that the column values display with a percent sign. The column name is aliased as 'Percent'.

Predicate

The predicate must be a boolean expression. Its purpose is to apply criteria for filtering objects out of the extent. Look at this query:

```
history:|bql:select timestamp, value from /weatherStation/outsideAirTemp
```

This query would retrieve the timestamp and value of all records in the specified history. That's often not a useful query and depending on how long the history has been collected, it may return a lot more data than we care to see. Instead, let's find all records where the value exceeds 80 degrees.

```
history:|bql:select timestamp, value from /weatherStation/outsideAirTemp where value > 80
```

By adding the "where" clause with "value > 80", all records with a value less than 80 are filtered out of the result. To learn more about BQL expressions, see **BQL Expressions**.

Having

The "having" clause must be a boolean expression. The having clause has the same semantics as in SQL. You can use the having clause to filter the results of your query based on aggregate functions. Consider this query:

```
select displayName, SUM(out.value) from control:NumericPoint having SUM(out.value) > 100
```

First, note that this query could return multiple rows since its projection contains both scalar columns ("displayName") and aggregate columns ("SUM(out.value)"). Each row will contain a distinct displayName, and the SUM of all the "out.value" values for the objects with that displayName. The HAVING clause will further restrict the result to only contain rows where the SUM of all the out.value values is greater than 100.

Note that if the above query had only asked for "SUM(out.value)" and did not ask for the displayName, there would only be one row in the result. It would contain the SUM of all the "out.value" values regardless of the object's displayName. It would not be very useful to include a HAVING clause in such a query.

Order By

The "order by" clause can be used to sort the results of the bql query. It also has similar semantics to SQL. You can order by a path expression, a column alias, or column position (using a 1-based index). Further, you can specify whether you want the ordering to be done in ascending (ASC) or descending (DESC) order. ASC is assumed if not specified. For example,

```
select displayName, slotPath as 'Path' from control:NumericPoint order by out.value, 1, 'Path'
DESC
```

Group By

BQL does not have a GROUP BY clause. If you mention *ANY* path expression in a query that contains aggregate functions, BQL implicitly defines a distinct grouping based on all the unique path expressions in your query. Consider:

```
select displayName, MAX(out.value) from control:NumericPoint where isWritablePoint
```

This query will cause the bql engine to define an implicit grouping based on the "displayName" and "isWritablePoint" values.

Simple Expressions

In some cases, it may be desirable to fetch a single value instead of a table of objects. You can can accomplish that with BQL by using a simple BQL expression.

```
slot:/a/b/c|bql:handle
```

Putting a simple path expression in the BQL ord, causes the expression to be evaluated relative to the base. Resolving this ord just returns the value of the expression. In this case the result is the handle of the component identified by "/a/b/c". Note: If you run this query in Workbench, you will get a "No views are accessible" error since the there are no views registered on the simple type "java.lang.String", which is the type of the "handle" path expression.

Beginning in Niagara 3.5, you can evaluate multiple expressions against the base and have the results returned in a table with a single row.

```
slot:/a/b/c|bql:{handle, out.value * 100, displayName + ' is my name'}
```

Each of the expressions in the list is evaluated against the component at "slot:/a/b/c". The result is a table with a single row with the result of evaluating each expression in its corresponding column.

BQL Paths

BQL paths are an important element of any BQL query. A path can be used to specify column content or to filter the rows in a query result. In all cases, a path is relative to the set of objects defined by the extent.

A path is a dot-separated list of fields. Consider the following example:

```
slot:/a/b|bql:select name, historyConfig.capacity from history:HistoryExt
```

This retrieves the name and configured capacity of all history extensions under "/a/b". The extent tells me that I am only looking for history extensions. The second column specifier tells me to look inside the historyConfig and extract the value of the "capacity" property. The same concept can be applied in the "where" clause.

```
slot:/a/b|bql:select name, out from control:NumericPoint where out.value > 50
```

In this case, the extent tells me that I am only looking for numeric points. The where clause looks at the "value" property of the "out" property of each numeric point in "/a/b" and only includes the ones that are greater than 50.

Presenting a list of all available fields in a path is not feasible. The fields that can be accessed in a path include all frozen and dynamic properties of any component or struct (given sufficient security permissions) plus many of the methods on the target type. The Bajadoc reference is the best place to find this information for a particular type.

A method is accessible via a path if it is public and returns a non-void value and takes either no parameters or only a Context as a parameter. Methods that match the "getX" pattern are handled specially. To access a getter from BQL, the "get" is dropped and the next letter is changed to lowercase resulting in the name of the desired value rather than the method name for getting it.

```
getX -> x
getCurrentTemperature -> currentTemperature
```

A few methods are used particularly often. "name" gets the slot name of a value on its parent. "parent" get the parent component. "parent" is useful because it allows you to look up the component tree.

```
slot:/foo/bar|bql:select parent.name, parent.slotPath from schedule:BooleanSchedule
```

This query finds the name and path of all containers that contain a BooleanSchedule.

For more examples, see **BQL Examples**.

For more information about expressions, see **BQL Expressions**.

Scalar and Aggregate Functions

BQL supports two types of function expressions: 1) scalar functions and 2) aggregate functions. Scalar functions operate on a single value and return a single value. In this respect they are similar to path expressions. Aggregate functions operate on a set of values, and return a single, summarizing value. BQL also supports the ability for programmers to create their own scalar and aggregate functions. In all cases, the syntax for calling a function is

```
(<type spec>.)<function name>(<parameter list>)
```

The type spec is only required when the function is not part of the built-in BQL library. This is described in more detail in the sections below.

Scalar Functions

BQL provides the following built-in scalar functions

- BBoolean slotExists(BString slotName): return true if an object has a slot with the given name.

- BBoolean propertyExists(BString propName): return true if an object has a property with the given name.
- BString substr(BString str, BNumber start, BNumber end): similar to Java substr() function.

```
select substr(displayName, 0, 1) from baja:Folder
select slotPath, displayName from baja:Component where slotExists('out')
```

The first query returns the first letter of all BFolders. The second query returns the slot path of every BComponent that has an 'out' slot.

User-defined Scalar Functions

In this example, we show how to create a new scalar function "strlen" that returns the length of a BString. To create a new scalar function you simply define a new `public static` method in one of your BObjects where the first parameter is a BObject (the target object to work with), and the rest of the parameters match the type of the parameters for your method. The return type of all BQL functions must be a BObject.

```
public BLib extends BObject {
  /** Define the strlen function */
  public static BInteger strlen(BObject target, BString str) {
    return BInteger.make(str.getString().length());
  }

  public static final Type TYPE = Sys.loadType(BBLib.class);
  public Type getType() { return TYPE; }
}
```

That's it! Pretty straight-forward. Assuming this function was in a module called "MyBql", here is how you could use it to get the displayName and its length for every BFolder (note the use of the BTypeSpec to call the function):

```
select displayName, MyBql:Lib.strlen(displayName) from baja:Folder
```

Aggregate Functions

BQL provides the following built-in aggregate functions:

1. COUNT(<expresion>): count the number of items in the result set. Supports special syntax COUNT(*).
2. MAX(<expression>): evaluates the expression for every item in the result set and returns the maximum value. The expression must evaluate to a BNumber or BStatusNumeric.
3. MIN(<expression>): evaluates the expression for every item in the result set and returns the minimum value. The expression must evaluate to a BNumber or BStatusNumeric.
4. SUM(<expression>): evaluates the expression for every item in the result set and returns the sum of all the values. The expression must evaluate to a BNumber or BStatusNumeric.
5. AVG(<expression>): evaluates the expression for every item in the result set and returns the average of all the values. The expression must evaluate to a BNumber or BStatusNumeric.

```
select MAX(out), MIN(out), AVG(out), SUM(out) from control:NumericWritable
select substr(displayName, 0, 1), COUNT(*) from baja:Folder
```

The first query returns the max, min, average, and sum of all the out properties of all control:NumericWritables. The resulting table will have a single row with four columns. The second query gets the first letter of every folder and then counts how many folders start with that letter.

User-defined Aggregate Functions

Note: The ability to create user-defined aggregate functions is still considered experimental. The steps to create aggregate functions may change in the future.

In this example we show how to create and implement the AVG() aggregate function provided by BQL. Creating an aggregate function is a two-step process. The process is outlined below, and then a code example is provided.

- Step 1: Create the aggregator class
  - Create a class that extends BObject and implements the "marker" javax.baja.bql.BIAggregator interface. This interface has no methods, it just serves to signal the BQL engine that the class has aggregator semantics.
  - By convention, you must have a `public void` method called "aggregate" with a single parameter that is the type you want to aggregate on. This method will be called for each object in the result set. This is where the aggregating should be done.
  - By convention, you must have a "public <Type>" method called "commit()" that returns the aggregate value. This will be called on your class when all the objects have been aggregated. This gives you a chance to do any further calculation before returning the result.
- Step 2: Declare the aggregator in one of your module's classes so that BQl can find it when given a BTypeSpec invocation of the aggregate function.
  - Declare a `public static final Type[] <function name>` in one of your module's classes. The <function name> is the actual function name that would be used in a bql statement. If you have multiple implementations of the aggregate function (perhaps to support different argument types), include them all in the array. The BQL engine will search the list of implementing classes until it finds one that implements an "aggregate(<type>)" method that matches the type of the current object.

Here is an implementation of AVG that supports averaging BNumbers and BStatusNumerics. This code example shows how to implement step 1 above.

```
public final class BAverage extends BObject implements BIAggregator {

  /** Aggregate a BNumber */
  public void aggregate(BNumber value) {
    ++count;
    sum += value.getDouble();
  }

  /** Only aggregates if the status is valid. Otherwise it is skipped */
  public void aggregate(BStatusNumeric value) {
    if (value.getStatus().isValid()) {
      ++count;
      sum += value.getValue();
    }
  }

  /** Calculate the average and return the result */
  public BDouble commit() {
    if (count == 0)
      return BDouble.NaN;
    else
      return BDouble.make(sum/count);
  }

  public static final Type TYPE = Sys.loadType(BAvg.class);
  public Type getType() { return TYPE; }

  private double sum;
  private long count;
}
```

In the scalar example above, we created a class "BLib" in the "MyBql" module to create the "strlen()" function. Here is how we can modify that class to define the AVG function we just created. This shows how to implement step 2 from the outline above.

```
public BLib extends BObject {
  /** Define the strlen function */
  public static BInteger strlen(BObject target, BString str) {
    return BInteger.make(str.getString().length());
  }

  // Declare the AVG aggregate function (step 2)
  public static final Type[] avg = { BAverage.TYPE };

  public static final Type TYPE = Sys.loadType(BBLib.class);
  public Type getType() { return TYPE; }
}
```

Note that the name of the aggregate function is determined by its declaration in step 2, it is ***NOT*** the name of the class that implements the aggregation logic. Also, aggregate names are case-insensitive. Here is how you would call your implementation of the average aggregate function (note the use of the BTypeSpec)

```
select MyBql:Lib.avg(out) from control:NumericWritable
```

BQL from Java

BQL query results can easily be displayed in a table or chart in a user interface. However, the results may also be examined in code using the Baja API. The result of a "select" query is always a BICollection. The items in the collection depend on the query. If the projection is omitted, the result is a collection of objects in the extent that matched the predicate requirements.

```
BOrd ord = BOrd.make("slot:/foo/bar|bql:select from control:NumericPoint");

BICollection result = (BICollection)ord.resolve(base).get();
Cursor c = result.cursor();
double total = 0d;
while (c.next())
{
  total += ((BNumericPoint)c.get()).getOut().getValue();
}
```

If the query has a projection, the result is a BITable and must be accessed that way to get the column data.

```
BOrd ord = BOrd.make("slot:/foo/bar|bql:select name, out.value from control:NumericPoint");

BITable result = (BITable)ord.resolve(base).get();
ColumnList columns = result.getColumns();
Column valueColumn = columns.get(1);
TableCursor c = (TableCursor)result.cursor();
double total = 0d;
while (c.next())
{
  total += ((BINumeric)c.get(valueColumn)).getNumeric();
}
```

Beginning in Niagara 3.5 you can perform BQL queries against unmounted components. This is useful when you are programmatically constructing component trees, and want to query the tree structure, but the components are not mounted in a station or bog. The example below illustrates how to do this.

```
 // NOTE: using setOut() for numeric writables because set() doesn't work when not mounted.
```

```
BFolder folder = new BFolder();
BNumericWritable nw1 = new BNumericWritable();
nw1.setOut(new BStatusNumeric(50.0));
folder.add("a", nw1);

nw1 = new BNumericWritable();
nw1.setOut(new BStatusNumeric(100.0));
folder.add("b", nw1);

String bql = "select sum(out.value) from control:NumericWritable";

// Create the unmounted OrdTarget using new "unmounted" factory method
OrdTarget target = OrdTarget.unmounted(folder);

// Query the unmounted folder to get the sum of all children
// control:NumericWritables out.value values.
BICollection coll = (BICollection)BqlQuery.make(bql).resolve(target).get();
```

BQL Expressions

Back to **BQL Overview**

BQL Expressions are used in the `where` clause of a BQL query to further qualify a result by narrowing the set of objects in the extent.

Operator Precedence

BQL supports the following set of operators ordered by precedence:

`!, not, -`	logical not, numeric negation
`*, /`	multiplication, division
`+, -`	addition, subtraction
`=, !=, >, >=, <, <=` `like, in`	comparisons
`and, or`	logical operators

Parentheses can be used to override the normal precedence.

Typed Literals

All primitive types and BSimple types can be expressed as literals in BQL. The syntax for primitives types is:

String - single quoted string
Example: 'This is a string literal'

number - a numeric value, unquoted
Example: 10

boolean - true or false, unquoted
Example: true

enum - The enum type spec followed by the tag separated by a dot.
Example: alarm:SourceState.normal

Expressing other BSimple types in BQL is more verbose because a type specifier is required. The syntax for a BSimple value is the type spec (i.e. moduleName:typeName) followed by a string literal with the string encoding of the value (the result of encodeToString() for the type). Example: baja:RelTime '10000'

Baja types are expressed in BQL using the type spec. Any type spec that is not followed by a quoted string refers to the type itself.
Example: where out.type = baja:StatusNumeric

BQL Examples

Back to **BQL Overview**

This document is a collection of example queries that illustrate how to identify some common sets of data with BQL. While each example in this document only presents a single solution, keep in mind that in most cases there are several different ways get the same result.

All points

```
select slotPath, out from control:ControlPoint
```

The result is the slot path and output value of all control points. Since we specified "out" the result is the combination of value and status. If we wanted just the value, we would have used out.value. Or if we wanted value and status in separate columns we would have specified out.value and out.status.

All points in alarm

```
select slotPath, out from control:ControlPoint where status.alarm
```

The result is the slot path and output value of all control points currently in the alarm state. In the where clause, the path "status.alarm" evaluates to true if the alarm status bit is set and false otherwise. This mechanism can be used to check the state of any of the status bits. See BStatus for more information on status flags.

All points with "Meter" in their name

```
select slotPath, out from control:ControlPoint where name like '%Meter%'
```

The result is the slot path and output value of all points whose name includes the substring "Meter". BQL supports simple pattern matching. A '%' or '*' matches zero or more characters. A '_' matches exactly one character. The normal character matching is case sensitive.

All points with a totalizer extension

```
select parent.slotPath, total from control:NumericTotalizerExt
```

The result is the slot path of every point that has a totalizer extension and the total for each totalizer. Note that the extent is the set of all totalizers. To get the point path, we look at the parent of each object in the extent.

All current schedule output values

```
select slotPath, out from schedule:AbstractSchedule stop
```

The result is the slot path and output value of all schedules. Note the keyword "stop". The schedule component model makes the "stop" keyword necessary. All of the common schedule (BooleanSchedule, NumericSchedule, etc.) are actually composed of many more precise schedules. Without the "stop", the result would include all of the inner schedules in addition to the top level schedules that this query is actually looking for. The "stop" tells the query processor to stop the recursion when it reaches a component whose type matches the extent type.

All points overridden at priority level 8

```
select slotPath, out from control:IWritablePoint
  where activeLevel = control:PriorityLevel.level_8
```

The result is the slot path and output value of all writable points that are currently overridden at priority level 8. I know that every writable point is an instance of BIWritablePoint. All writable points provide access to their active level with a method called getActiveLevel(). Following the pattern for translating method names to BQL fields, I can access the active level on writable points

using "activeLevel". In this case I know that active level is represented by a PriorityLevel enum. The level 8 value of the enum is specified by "control:PriorityLevel.level_8".

All points with units of degrees fahrenheit

```
select slotPath from control:NumericPoint
  where facets.units.unitName = 'fahrenheit'
```

The key to this query is understanding how units are associated with a point. All control points have facets. For numeric points, the units are defined as a facet. So facets.units gets the units for the point. BUnit has a method called getUnitName() so "unitName" gets the result of that method.

All points linked to a specific schedule

```
select targetComponent.slotPath from baja:Link
  where sourceSlotName = 'out' and
        sourceComponent.slotPath = 'slot:/app/MainSchedule'
```

This one is tricky. Because links are dynamic, they do not have a fixed name that we can search for. There is also no way to access just the links to a schedule output from BQL. Instead we have to look at all of the links and check the endpoints. So the extent is all links. Then we check for a source slot of "out". Finally we check the source slot path.

All points that generate alarms of a specific class

```
select parent.slotPath from alarm:AlarmSourceExt where alarmClass = 'hvac'
```

The result is the slot path of all control points that generate alarms for the "hvac" alarm class. The extent is all alarm source extensions. We find the extensions that specify "hvac" for the alarm class and get the parent slot path from those. The parent of an alarm source extension is always a control point.

All points with a history extension

```
select parent.slotPath from history:HistoryExt
```

This one is simple. We find all of the history extensions by using history:HistoryExt as the extent. Then we just get the slot path of the parent. The parent of a history extension is always a control point.

All points that collect a history with a capacity greater than 1000 records.

```
select parent.slotPath, historyConfig.capacity from history:HistoryExt
  where historyConfig.capacity.isUnlimited or
        historyConfig.capacity.maxRecords > 1000
```

For this query you have to understand how history extensions are configured. The capacity is a property of HistoryConfig. However, Capacity is not a simple numeric value. To exceed 1000 records of capacity, the configured capacity may either be unlimited or limited to a value greater than 1000. So first we check for unlimited and then we check for a limit of more than 1000 records.

The number of unacked alarms in all alarm classes

```
select name, unackedAlarmCount from alarm:AlarmClass
```

This query just looks at all of the alarm classes and for each one returns the name and the unackedAlarmCount. In this case, it will be much more efficient to narrow the search by making the alarm service be the query base. All alarm classes must be children of the AlarmService. So it is much better to only search the AlarmService container.

```
slot:/Services/Alarm|bql:select name, numberOfUnackedAlarms from alarm:AlarmClass
```

Driver Framework

Overview

The driver framework provides a common model for abstracting how information is imported and exported from the station VM. The model is built upon the following concepts

- `BDeviceNetwork`: This models a physical or logical network of devices.
- `BDevice`: This models a physical or logical device such as a fieldbus device or an IP host.
- `BDeviceExt`: This models a functional integration at the device level which imports and/or exports a specific type of information such a points, histories, alarms, or schedules.
- `BNetworkExt`: This models an functional extension at the network level.

Driver Hierarchy

Drivers are always structured according to a fixed slot hierarchy as illustrated the Driver Hierarchy Diagram:

- **DriverContainer**: Typically all drivers are located in this folder directly under the station root.
- **DeviceNetwork**: Models the specific driver's protocol stack.
- **DeviceFolder**: Zero or more levels of DeviceFolder can be used to organize the driver's Devices.
- **Device**: Devices model the physical or logical device of the driver. Devices are descendents of the DeviceNetwork either as direct children or inside DeviceFolders.
- **DeviceExt**: DeviceExts are always direct children of Devices, typically declared as frozen slots.

Within each DeviceExt, there is usually a well defined hierarchy. For example the PointDeviceExt follows a similar model with PointDeviceExt, PointFolders, ControlPoints, and ProxyExt.

Status

A key function of the driver framework is providing normalized management of status. The follows semantics are defined for status flags:

- **Disabled**: the user manually disabled the driver component
- **Fault**: a configuration, hardware, or software error is detected
- **Down**: a communication error has occurred
- **Stale**: a situation has occurred (such as elapsed time since a read) which renders the current value untrustworthy

The driver framework provides a standard mechanism to manage each of these status flags. A component is disabled when a user manually sets the enabled property to false. Disable automatically propagates down the tree. For example setting the network level disabled automatically sets all devices and points under it disabled.

The fault status is typically a merge of multiple fault situations. The driver framework does its own fault detection to detect fatal faults. Fatal faults typically occur because a device or component has been placed inside the wrong container (such as putting a ModbusDevice under a LonworksNetwork). Licensing failures can also trigger fatal faults. Driver developers can set their own fault conditions in networks and devices using the `configFail()` and `configOk()` methods. A `faultCause` method provides a short description of why a component is in fault. Fault conditions automaticlly propagate down the tree.

The down status indicates a communication failure at the network or device level. Down status is managed by the ping APIs using `pingFail()` and `pingOk()`. Ping status is maintained in the health property. The driver framework includes a `PingMonitor` which automatically pings devices on a periodic basis to check their health. The `PingMonitor` can generate alarms if it detects a device has gone down.

DeviceExts

The following standard device extensions provide a framework for working specific types of data:

- Point: For reading and writing proxy points.
- History: For importing and exporting histories.

- Alarm: For routing incoming and outgoing alarms.
- Schedule: Used to perform master/slave scheduling.

User Interfaces

The driver framework provides a comprehensive set of APIs for building tools for managing configuration and learns based on the AbstractManager API. Also see the Driver Learn illustration.

Point Devicelet Framework

Overview

The `javax.baja.driver.point` API is used to perform point IO with logical or physical control points. Drivers use the standard control points found in the `control` module. But each driver provides a specialization of `BProxyExt` for driver specific addressing, tuning, and IO.

Refer to Architecture - Driver Hierarchy for an illustration of the component slot hierarchy.

Refer to Architecture - ProxyExt for an illustration of the design.

Point Modes

There are three modes which a proxy point may operate in:

- **Readonly**: These points are read from the device, but never written.
- **ReadWrite**: These are points which the driver can both read from and write to.
- **Writeonly**: These are points which the driver can write to, but cannot read.

A ProxyExt must indicate which mode it is operating by overriding the `getMode()` method

.

Proxy Ext

The `ProxyExt` component contains two properties used for managing read and write values.

The `readValue` property indicates the last value read from the device. For writeonly points this is the last value successfully written. This value is used to feed the parent point's extensions and out property. If numeric, it is in device units.

The `writeValue` property stores the value currently desired to be written to the device. If numeric, it is in device units.

Framework to Driver Callbacks

Driver developers have three callbacks which should be used to manage reads and writes:

- `ProxyExt.readSubscribed()`: This callback is made when the point enters the subscribed state. This is an indication to the driver that something is now interested in this point. Drivers should begin polling or register for changes.
- `ProxyExt.readUnsubscribed()`: This callback is made when the point enters the unsubscribed state. This is an indication to the driver that no one is interested in the point's current value anymore. Drivers should cease polling or unregister for changes.
- `ProxyExt.write()`: This callback is made when the framework determines that a point should be written. The tuning policy is used to manage write scheduling.

Note: All three callbacks should be handled quickly and should never perform IO on the callers thread. Instead drivers should use queues and asynchronous threads to perform the actual IO.

Driver to Framework Callbacks

The `ProxyExt` contains a standard API which the driver should call once a read or write operation has been attempted.

If a read operation completes successfully then `readOk()` method should be called with the value read. If the read fails then call the `readFail()` method.

If a write operation completes successfully then the `writeOk()` method should be called with the value written. If the write fails for any reason then call `writeFail()`.

Tuning

All ProxyExts contain a Tuning property that manages how read and writes are tuned. All drivers which implement proxy points should create a "tuningPolicies" property of type TuningPolicyMap on their DeviceNetwork. The Tuning structure on each ProxyExt identifies its TuningPolicy within the network by slot name. TuningPolicies allow users to configure which state transitions result in a

`write()` callback. TuningPolicies may also be used to setup a minWriteTime to throttle writes and a maxWriteTime to do rewrites.

Utilities

The driver framework provides a suite of APIs to aid developers in building their drivers:

- `BPollScheduler`: This is a prebuild component that manages polling the points using a set of configurable buckets. To use this feature have your `ProxyExt` implement the `BIPollable` interface.
- `ByteBuffer`: This class provides a wealth of methods when working with byte buffers such as reading and writing integers using big or little endian.

History Devicelet Framework

Overview

Refer to the javax.baja.driver.history API.

History device extensions manage exporting and importing histories (data logs) to and from remote devices for archiving purposes. For more information on Niagara histories, refer to the History documentation.

The BHistoryDeviceExt component is the container for archive descriptors which specify the details for importing/ exporting histories. A concrete implementation of this component can be placed under a device (concrete implementation of BDevice) to specify the export/import behavior of histories to and from the device. The actual descriptions of each history export/import are contained in a subclass of BArchiveDescriptor which supplies the unique History Id for the history exported/imported. Since it is a BDescriptor, among other things it supplies the execution time for performing the export/import. Two subclasses of `BArchiveDescriptor` are available: BHistoryExport is used for exporting or pushing a history to a remote device (referred to as a history export descriptor), and BHistoryImport is used for importing or pulling a history from a remote device (referred to as a history import descriptor). Currently these are the only two options, or active history descriptors. At present there are no passive history descriptors (i.e. history export*ed* descriptor or history import*ed* descriptor). Also, in the concrete Niagara Driver implementation, the code prevents a history export from occuring when there already exists a history import for a matching history id.

The BHistoryNetworkExt component manages network level functions for the history transfers. Its primary purpose is to be the container of the configuration rules (BConfigRules) that specify how the configuration of a history should be changed when a history is pushed (exported) into a Niagara station. Configuration rules are applied when an exported history is created. Changing a rule has no effect on existing histories. A `BConfigRule` entry has two String parameters used for matching a pushed history's device and history name, and once a match is found (the configuration rules are iterated in slot order, and the first match will be used), any override rules (properties) will be used in place of the corresponding properties on the incoming history's configuration (BHistoryConfig). For example, if you wanted to increase the history capacity on a history that has been received from an export for archiving purposes, you could supply an override property on a configuration rule to increase the capacity.

Alarm Devicelet Framework

The BAlarmDeviceExt handles the sending an receiving of alarms to and from remote devices. It is both and alarm source, implementing BIRemoteAlarmSource, and an alarm recipient, implementing BIRemoteAlarmRecipient.

Receiving Alarms

BAlarmDeviceExt is used for receiving alarms from a remote device. The BAlarmDeviceExt should be used as the source for all incoming alarms. If more detail is needed about the actual source, the BAlarmRecord.SOURCE_NAME or additional fields in the BAlarmRecord's alarmData can be used. Alarm Ack Request will be routed back to the BAlarmDeviceExt when it is set as the source.

In Niagara Offnormal and Normal alarms are not two separate alarms as is found in some systems. In Niagara Offnormal and Normal are two states of the same alarm. This is important to keep in mind is not using the AlarmSupport class as each offnormal alarm generated will need it's source state set to Normal when it's source goes back to the normal state.

Sending Alarms

Sending alarms from the Niagara system to a remote device is accomplished by implmenting a BAlarmRecipient. The BAlarmRecipient's handleAlarm method should route alarms from the Niagara system to the remote device and the originating source. The actual sending of alarms to the device network should be done on a separate thread so as to not block the control engine thread. The DeviceExt should not attempt to send alarms to Devices which are down or disabled.

Schedule Device Extensions

Overview

Schedule device extensions manage remote schedule synchonization. A subordinate schedule is a read-only copy of a supervisor schedule. Subordinate schedules must be children of the schedule device extension.

Refer to the javax.baja.schedule.driver API.

BScheduleDeviceExt

Container of supervisor schedule export descriptors and subordinate schedules.

Subscription

At a random time after station startup and within the subscribeWindow property value, all subordinate schedules who have not communicated with their supervisor will have their execute action invoked. For drivers where remote supervisors do not persist information about local subordinates, the subscribe window should be some small value rather than the default of a day.

Retries

Periodically the execute action of all BScheduleExports and BScheduleImportExts who are in fault is invoked. The retry interval is controled by the retryTrigger property.

Subclasses

- Implement makeExport(String supervisorId) to create BScheduleExport objects for incoming subscription requests from remote subordinates.
- Implement makeImportExt() to create the schedule extension for new subordinate schedules.
- Can call processImport() to handle requests from remote subordinates.
- Can call processExport() to handle updates from remote supervisors.

BScheduleExport

Maps a local supervisor to a remote subordinate. Will be a child of a BScheduleDeviceExt.

Execution

The execute action is where the the local supervisor schedule configuration is sent to the remote subordinate. It is only invoked if the local supervisor schedule has been modified since the last time it was sent to the remote subordinate. The executionTime property controls when the local supervisor version is compared to the remote subordinate.

Subclasses

- Implement doExecute() to upload the supervisor schedule configuration.
- Implement postExecute() to enqueue the execute action on an async thread.
- Always call getExportableSchedule() before encoding a schedule for transmission. This inlines schedule references.

BScheduleImportExt

Maps a local subordinate to a remote supervisor. Will be a child of the subordinate schedule.

Execution

The execute action is where the local subordinate makes a request to the remote supervisor for a configuration update. The executionTime property controls when execute is invoked but it is turned off by default. Since BScheduleImportExt.execute will always result in a message to the remote supervisor, it is more efficient to have the supervisor push changes only when necessary.

When the schedule device extension performs subscription, it is simply invoking the execute action on BScheduleImportExt.

Subclasses

- Implement doExecute() to download the supervisor schedule configuration.
- Implement postExecute() to enqueue the execute action on an async thread.
- Can call processExport() to handle configuration updates from the remote supervisor.

BScheduleExportManager

This is the manager view for local supervisor schedules. This is a convenience and can be ignored.

Subclasses

- Subclass ScheduleExportModel to add MgrColumns for properties added to your BScheduleExport.
- Override makeModel() to return your new model.
- Make the manager an agent on your schedule device extenstion.

BScheduleImportManager

This is the manager view for local subordinate schedules. This is a convenience and can be ignored.

Subclasses

- Subclass ScheduleImportModel to add MgrColumns for properties added to your BScheduleImportExt.
- Override makeModel() to return your new model.
- Make the manager an agent on your schedule device extenstion.

Basic Driver

Overview

Refer to the Basic Driver API.

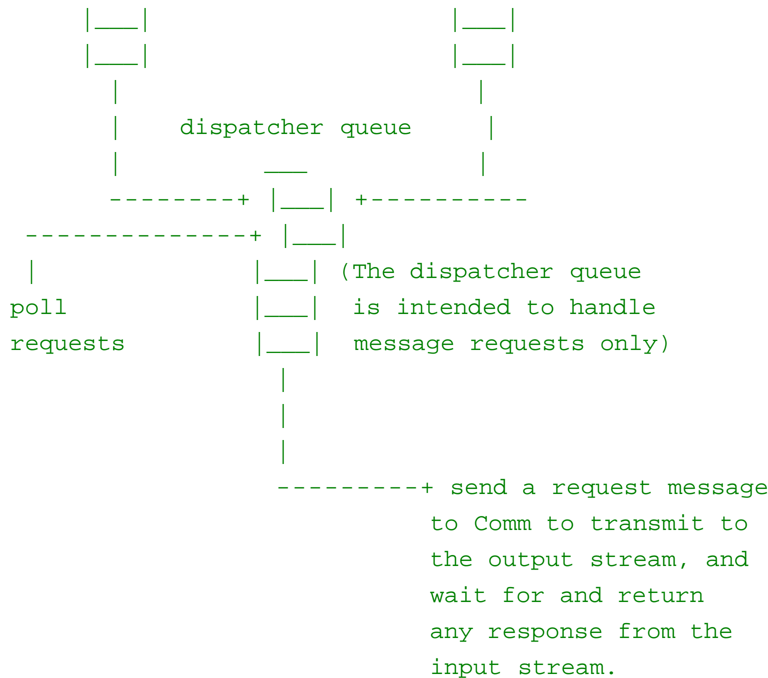
This package provides some basic classes that may be useful to developers building a new driver (i.e. field bus driver). These classes can be used (or subclassed) to provide some basic driver functionality, such as worker (queue) management, basic poll schedule handling, basic messages and management of these basic messages through request/response transactions (as well as unsolicited message handling), etc. It also provides a serial implementation (com.tridium.basicdriver.serial) which can be subclassed by drivers that use a serial port for communication. Here is an overview of basicDriver's structure:

```
                BBasicNetwork
               /   /    |    \
              /   /     |     \
  BBasicDevices  /      |   Worker Queues (BBasicWorkers)
                /       |    - dispatcher (used for
               /        |      synchronizing access to Comm)
BBasicPollScheduler     |     - worker (for posting async
                        |      operations, such as learns)
                        |    - write worker (for posting
                        |      async coalescing operations,
                        |      such as writes)
                        |
                      Comm
                     /  |  \
                    /   |   \
       CommReceiver     |   CommTransactionManager
       CommTransmitter  |    - CommTransactions
                        |
                        |
          UnsolicitedMessageListeners
       (registered if needed by network)
```

The abstract class BBasicNetwork is the root component of basicDriver. It is the base container for BBasicDevice objects, and it provides a basic poll scheduler where objects implementing the BIBasicPollable interface can register to be polled (i.e. points, devices). It also provides three worker threads (queues) for handling asynchonous operations and synchronization of request messages to the Comm for transmission to the output stream (the following outlines the INTENDED use of these worker queues):

Asynchronous operations should be posted onto either the worker queue or write worker queue (coalescing). Write operations should always go to the write worker queue so they will be coalesced. Most other asynchronous operations, such as learns, should be posted to the worker queue to keep the write worker queue free for write operations. As these async operations are processed (dequeued), they should post any necessary message requests to the dispatcher queue, which synchronizes access to the `Comm` (`Comm` is ultimately responsible for sending the request message to the output stream via the `CommTransmitter` and receiving the response message from the input stream via the `CommReceiver`). Other threads may also post directly to the dispatcher queue (for example, the poll thread can post poll message requests directly to the dispatcher queue).

```
    worker queue          write worker queue
       ___                     ___
      |___|                   |___|
      |___|                   |___|
      |___|                   |___|
```



Supporting Classes

`BBasicNetwork` also handles initialization, starting, and stopping the `Comm`, or communication handler. `Comm` is used to manage request/response message transactions for the network, handles the interaction between the low-level transmitter and receiver, and routes any unsolicited received messages to the appropriate listener. `Comm` uses the following supporting classes to accomplish its tasks:

- CommTransactionManager: provides a pool of CommTransaction objects that are used for request/response message matching. Matching a request message to a response message is determined through an `Object` tag on the `Message` (discussed below).
- CommReceiver: an abstract class implementing `Runnable` which handles receiving and forming `ReceivedMessage`s from the input stream. Subclasses must override the `receive()` abstract method to read and return a complete `ReceivedMessage`. `CommReceiver` will loop and continuously call `receive()` in order to receive messages. Once a complete `ReceivedMessage` is received, this class routes the `ReceivedMessage` back up to the `Comm` for further processing. The returned `ReceivedMessage` may also need to contain data for request/response message matching (tag data) and unsolicited message listener processing (unsolicited listener code).
- CommTransmitter: provides access and synchronization for writing `Message`s (and/or bytes) to the output stream.
- UnsolicitedMessageListener: `Comm` can store a list of objects implementing this interface in order to process unsolicited received messages. `UnsolicitedMessageListener` objects can be registered to the `Comm` with an unsolicited listener code key. Then when a `ReceivedMessage` is received and determined to be unsolicited, it can match the unsolicited listener code to determine which `UnsolicitedMessageListener` instance should handle the `ReceivedMessage`.
- MessageListener: This is a helper interface that should be implemented by objects that wish to receive a response `Message`. When using the `sendAsync()` or `sendAsyncWrite()` convenience methods of `BBasicNetwork`, they require a parameter of type `MessageListener` in order to determine where to route the response `Message`.

Messages

The com.tridium.basicdriver.message package contains classes useful for building driver messages (using the `Message` abstract class), allowing these `Message`s to be written to the output stream, and formatting a response received (`ReceivedMessage`) into a proper `Message`.

- Message: an abstract class for wrapping a driver message and providing some methods necessary for handling a response to this message. At a minimum, subclasses will need to provide the implementation for writing the message to the output stream and determine how a response (`ReceivedMessage`) should be interpreted and formed into a `Message`.

- ReceivedMessage: an abstract class for wrapping a received driver message and providing some methods for determining if it is unsolicited and/or the unsolicited listener code to use for finding the correct `UnsolicitedMessageListener` if the message is determined to be unsolicited. Subclasses should provide a means to serve the appropriate data to form a complete `Message`.

Utility Classes

The com.tridium.basicdriver.util package contains utility classes useful to most drivers.

- BasicException: an extension of BajaRuntimeException, a `BasicException` can be thrown when an error occurs in the driver.
- BBasicWorker: an extension of BWorker, it manages a basic worker thread for a queue. Used by the `BBasicNetwork` for the asynchronous worker.
- BBasicCoalescingWorker: an extension of `BBasicWorker`, it manages a basic worker thread for a coalescing queue. Used by the `BBasicNetwork` for the asynchronous write worker.
- BBasicPollScheduler: an extension of BPollScheduler, it handles subscribing, unsubscribing, and polling of `BIBasicPollable` objects.
- BIBasicPollable: an extension of BIPollable, this interface should be implemented by any objects that wish to register to receive poll requests from the `BBasicPollScheduler`. Subclasses of basicDriver can use this to poll any devices, points, etc. as needed.

Serial Driver

The com.tridium.basicdriver.serial package contains classes useful to most serial drivers (with the communication handler, `Comm`, at the network level).

- BSerialNetwork: an extension of `BBasicNetwork` that supports serial communication on a single configurable serial port. This abstract class can be subclassed to provide a frozen property of type BSerialHelper. This property, called 'Serial Port Config', provides an end user the ability to configure a serial port and its settings (i.e. baud rate, data bits, etc.) to use for communication with devices on the serial network.
- SerialComm: an extension of `Comm` that handles opening the user selected serial port as well as the input and output streams to that port. It is used by the `BSerialNetwork` to handle synchronization of the serial communication.

BACnet Driver

Overview

The Niagara AX BACnet driver provides both client and server side BACnet functionality. On the server side, the Niagara station is represented as a BACnet device on the network. Certain objects in Niagara can be exposed as BACnet Objects. Niagara will respond to BACnet service requests for these objects, according to the BACnet specification. On the client side, Niagara can represent other BACnet devices in the Framework. Properties of BACnet objects can be brought into Niagara as BACnet Proxy Points. In addition, the BACnet driver provides client side schedule and trend log access. The BACnet objects can also be viewed as a whole, using the Config views. Both client-side and server-side alarm support is provided, using the intrinsic alarming mechanism. The basic components in the BACnet driver are

- `BBacnetNetwork`: This represents the BACnet network in Niagara.
- `BLocalBacnetDevice`: This represents Niagara as a BACnet device.
- `BBacnetDevice`: This models a remote BACnet device.

Server

The server side functionality of the driver is accomplished by using **export descriptors** to map Niagara objects as BACnet Objects. The Local BACnet Device contains an export table from where all of the export descriptors are managed. The `javax.baja.bacnet.export` package contains the standard export descriptors. The base interface for an export descriptor, which must be implemented by all export descriptors, is `BIBacnetServerObject`. This contains the methods that are used by the comm stack and export mechanisms to access the BACnet Object properties of whatever Niagara object is being exported. The primary classes implementing this interface are

- `BBacnetPointDescriptor` and its subclasses - for exporting control points.
- `BBacnetScheduleDescriptor` and its subclasses - for exporting schedules.
- `BBacnetTrendLogDescriptor` and `BBacnetNiagaraHistoryDescriptor` - for exporting native BACnet trend logs and Niagara histories, respectively.
- `BBacnetFileDescriptor` - for exporting file system files.
- `BBacnetNotificationClassDescriptor` - for exporting Niagara BAlarmClasses as Notification Class objects.

Wherever a BACnet property is available directly from the exported Niagara object, this property is used. In some cases, a BACnet-required property is not available on the Niagara object being exported. In those cases, the property is defined within the export descriptor itself.

To export an object, the Bacnet Export Manager is used. A BQL query is made against the station to find components of a particular type, and the results are displayed. When a decision is made to add an export descriptor for a particular component, the registry is searched for export descriptors that are registered as agents on the component's Type. If any are found, these are presented to the user in the Add dialog.

For accepting writes, the BACnet driver requires that a BACnet user be defined in the User Service. The password for this user is not important, except for Device Management functions such as DeviceCommunicationControl and ReinitializeDevice. The permissions assigned for this user define what level of access is allowed for BACnet devices. Reads are always allowed; writes and modifications (such as AddListElement) are governed by the permissions of the BACnet user. If no BACnet user is defined, writes are not allowed.

The main area where the server side of the BACnet driver is extensible is through the creation of new export descriptor types. To create export descriptors for object types that are not currently exportable (such as a String Point), you simply need to create a class that implements `BIBacnetServerObject`. You may find that you want to subclass one of the base export descriptor classes mentioned above, or you may find it easier to create your own, using these classes as a guide.

Client

The client side functionality of the driver is accomplished with the `BBacnetDevice` and its device extensions. There are extensions for each of the normalized models

- `BBacnetPointDeviceExt` - for modeling properties of BACnet objects into Niagara control points.
- `BBacnetScheduleDeviceExt` - for representing BACnet Schedules as Niagara schedules for monitor or control.

- `BBacnetHistoryDeviceExt` - for representing BACnet Trend Logs as Niagara histories for configuration and archiving.
- `BBacnetAlarmDeviceExt` - for managing BACnet alarms from the device.
- `BBacnetConfigDeviceExt` - for viewing and modifying BACnet Objects in their native model - as an entire object, rather than by individual properties.

BACnet Proxy Points are configured by using a `BBacnetProxyExt`. There are four subclasses of this, one for each type of Niagara control point. The extensions are polymorphic, in that they know how to convert data from any of the primitive data types to the data type of their parent point. Any proxy point can be written to if it is of the proper type. The BACnet proxy extensions manage writes for both priority-array and non-prioritized points.

BACnet client-side Scheduling can be accomplished in two ways

- `BBacnetScheduleExport` - This descriptor is used when Niagara is the supervisor, driving the schedule in the device. It contains the object identifier of the remote schedule, and the ord to the Niagara schedule that is to be the source of scheduling data. At configurable times this data is written down to the remote schedule.
- `BBacnetScheduleImportExt` - This extension is used when the remote schedule is the source of data, and Niagara is simply reading scheduling information from the device. The schedule is queried at configurable times to update the Niagara schedule.

BACnet client-side Trending is accomplished by using the `BBacnetHistoryImport`. This descriptor periodically archives data from the Trend Log object in the remote device for storage by the Niagara station.

The operation of BACnet objects is sometimes easier to understand when the object is viewed as a whole, with all of its properties viewed toegether. For this reason, the Config device extension is provided. This allows you to view, for example, all of the properties of an Analog Input object together, without having to create proxy points for all of them. The expected use case is initial configuration or commissioning. The base object for representing BACnet Objects is `BBacnetObject`. Specific subclasses for BACnet standard object types exist in `javax.baja.bacnet.config`.

The main areas where the client side of the BACnet driver is extensible are

1. **`BBacnetDevice`**. For specialized device behavior, the `BBacnetDevice` can be subclassed. This is not for adding additional BACnet properties; the device object properties are contained in the `BBacnetDeviceObject`. Each `BBacnetDevice` has an enumeration list which contains all of the extensions known to that device. Specific device classes might have preconfigured entries for these enumerations, that allow it to better interpret and represent proprietary enumeration values received from this device.
2. **`BBacnetObject`**. For specialized object types, such as a representation of a proprietary object type, the `BBacnetObject` class should be subclassed. This includes any specific device object properties, which would be contained in a subclass of `BBacnetDeviceObject`.
3. proprietary **data types**. If any proprietary data types are created, they can be modelled corresponding to the data types in `javax.baja.bacnet.datatypes`. Primitive data types are generally modelled as simples. Constructed data types are generally modelled as a subclass of BComplex. The data type must implement `BIBacnetDataType`.
4. proprietary **enumerations**. Proprietary enumerations can also be created. If a property in an object is of an extensible enumeration, it should be modelled as a dynamic enum whose range is defined by the specified frozen enum. Examples of both extensible and non-extensible enumerations exist in `javax.baja.bacnet.enum`.

For additional information, refer to the BACnet API.

Lonworks Driver

Overview

The Lonworks API provides the means to model lonwork networks and devices for configuration and run time control. A network is a collection of connected lonworks devices and routers.

Basic components

- `BLonNetwork`: Is the top level container for `BLonDevices`. It provides manager views for commissioning, binding and trouble shooting.
- `BLonDevice`: Provides a database model for lonDevices to facilitate configuration and access to run time data.
- `BLonProxyExt`: Customizes proxy points for data elements on lonDevices. Proxy points provide the mechanism to interface point data in devices with Niagara control logic and graphics

Misc components

- `BLonRouter`: Contains the database model needed for network management of lonworks router.

LonDevice

A `BLonDevice` contains `BDeviceData` and the means to manage a collection of `BLonComponents` and `BMessageTags`.

`BLonComponents` are database representation of specific components in a device. They contain one or more data elements (see LonDataModel below) and specific config information. There are three types: `BNetworkVariable`, `BNetworkConfig`, `BConfigParameter`.

`BMessageTags` are only for linking. There is no behavior implemented in massage tags in the station.

`BLonDevice` is an abstract class and is the root class for all lonworks devices. There are two flavors of `BLonDevice` implemented in the lonworks drivers:

- `BLocalLonDevice` is a final class which provides the means to manage the local neuron. It is a frozen slot on `BLonNetwork`.
- `BDynamicDevice` provides support for dynamically building the devices data and `BLonComponents`. There are two actions to accomplish this: **learnNv** uses the self documentation in the device, **importXLon** uses an xml file containing a representation of the device.

Lon Data Model

Each `BLonComponent` contains one or more data elements which mirror the data structure of the component in the device. These data elements are managed by `BLonData` which handles conversion between the database representation and the devices binary format. `BLonData` can also contain other `BLonData` components allowing for nested data types.

NOTE: `BLonData` has been folded into `BLonComponent`. The effect is to place the data elements at the same tree level as the LonComponent config properties. This was done to improve efficiency in the workbench. The getter settor methods in `BLonComponent` access LonData as though it were contained by the LonComponent. The `getData()` method will return the `BLonComponent` as a `BLonData`. The `setData()` method will replace the current data elements with the new elements passed in data argument.

Each data element is modeled as a `BLonPrimitive`. There are `BLonPrimitives` for each primitive datatype.

- `BLonBoolean` models a boolean element
- `BLonEnum` models a enumeration element
- `BLonFloat` models a numeric element
- `BLonString` models a string element

Special element types

- `BLonInteger` models a numeric element when full 32 bits of data is needed

- BLonByteArray is used in special cases to model data when can not be meaningfully modeled as primitive elements
- BLonSimple is used in special cases to model data which has been representated by a simple. The simple must implement BILonNetworkSimple .

Proxy points

Proxy points are standard Niagara control points used to access data elements in foreign devices. Proxy points have a driver specific ProxyExt that handles addressing data elements in a specific device and data conversion needed to present the data in a normalized format. The inputs and outputs of proxies can be linked to other control logic or graphical points.

A BLonProxyExt in a Proxy point makes it a lonworks proxy point. There are different BLonProxyExts for each primitive data type. These can be seen in javax.baja.lonworks.proxy.

Lon Proxy Points are managed by LonPointManager which is a view on the points container in each BLonDevice.

Network Management

Implements a set of standard lonworks network management functions. The user has access to these functions through the following manager view.

- DeviceManager - provides support for discovering and adding lonwork devices to the database, for managing device addresses, and downloading standard applications to devices.
- RouterManager - provides support for discovering and adding lonwork routers to the database, and for managing device addresses
- LinkManagar - provides means to manage link types and bind links.
- LonUtiliesManager - provides a set of utilities useful for managing a lon network

LonComm

The lonworks communication stack can be accessed through a call to BLonNetwork.lonComm(). LonComm is provides APIs which allow the user to send LonMessages with one of the LonTalk service types (unackowledged, acknowledged, unackowledged repeat, request response).

LonComm also provides a means to receive unsolicited messages by registering a LonListener for a specifed message type from an optional subnetNode address.

LonMessage

LonMessage is the base class for all messages passed to/from LonComm APIs.

Users should subclass LonMessage if they wish to create a new explicit message type.

A set of LonTalk defined messages is provide in com/tridium/lonworks/netmessages. The definition of these message is found in Neuron Chip Data Book Appendix B, Lonworks Router User's Guide, and EIA/CEA-709.1-B.

Mapping ProgramId to Device Representation

A mechanism is provided to associate an xml or class file with a particular device type. The device type is identifyed by its ProgramId as described in LonMark Application-Layer Interoperability Guidelines. The association is created by putting "def" entries in a modules module-include.xml file. This association is used during the learn process to determine the appropriate database entity for discovered devices.

A "def" entry consists of name and value attribute. The name has the formate "lonworks:programId" where programId is the devices ProgramId represented as 8 hex encoded bytes with leading zeros and <space> delimiter. Multiple mappings are allowed for the same programId. Any nibble can be replaced with an '*' to indicate a range of programIds mapped to the same object. The value field can reference a class or xml file.

The formate for a class is cl=module:cname. The module is the niagara module containing the class and the cname is the name as defined in the module-include.xml for that module. The class must be a sub class of BLonDevice or BDynamicDevice.

The formate for an xml file is xml=module/xname. The module is the niagara module containing the xml file and the xname is the

name of the file containing the device representation. The xml file formate is described in Lon Markup Language.

Examples of def entries in lon device module-include.xml file.

```
<defs>
  <def name="lonworks.80 00 0c 50 3c 03 04 17"  value="cl=lonHoneywell:Q7300"  />
  <def name="lonworks.80 00 16 50 0a 04 04 0a"  value="xml=lonSiebe/Mnlrv3.lnml"/>
  <def name="lonworks.80 00 8e 10 0a 04 0* **"  value="xml=lonCompany/dev.lnml"/>
</defs>
```

For further information refer to the Lonworks API.

Lon Markup Language

Overview

This document defines the content of a lon XML interface file. Basic syntax: The xml interface file represents a collection of objects. These objects may contain predefined elements and/or undefined elements. LonDevice contains the predefined element deviceData and any number of undefined NetworkVariables. As a general rule, elements which are not a defined element of the parent must have a type attribute. Defined elements must provide a value "v" attribute or contain defined elements.

```
<name  type="XLonXmlType">
<name  v="value">
```

Example:

```
<T7300h type="XLonDevice">
 <!-- Defined element deviceData with no type specified -- >
 <deviceData
  <!-- Defined element with value -- >
  <programID v="80 0 c 50 3c 3 4 17"/>
  . . .
 </deviceData>

 <!-- Undefined element with specified type -- >
 <nviRequest type="XNetworkVariable">
  <index v="0"/>
 </nviRequest>
<T7300h type="XLonDevice">
```

The set of valid LonXmlTypes are: XLonXMLInterfaceFile, XLonDevice, XEnumDef, XTypeDef, XNetworkVariable, XNetworkConfig, XConfigProperty, XMessageTag

LonXMLInterfaceFile

The root type is LonXMLInterfaceFile. It may contain EnumDefs, TypeDefs, and LonDevices. It may also reference other LonXMLInterfaceFiles to allow for EnumDefs, and TypeDefs to be shared. The file attribute indicates the element is an included file

```
<!-- Example with reference to other interface files. -->
<T7300h type="XLonXMLInterfaceFile">
 <HwTherm file="datatypes\HwTherm.lnml"/>
 <HwCommon file="datatypes\HwCommon.lnml"/>
 <T7300h type="XlonDevice">
</T7300h>

<!-- Example with enumDefs and typeDefs included in single file. -->
<T7300h type="XLonXMLInterfaceFile">
 <HwThermAlarmEnum type="XenumDef"> . . . </HwThermAlarmEnum>
 <HwThermAlarm type="XTypeDef"> . . . </HwThermAlarm>
 <T7300h type="XLonDevice"> . . . </T7300h>
</T7300h>
```

TypeDefs

EnumDefs and TypeDefs elements are needed to define the data portion of nvs, ncis, and config properties. An EnumDef contains a set of tag/id pairs where the name of the element is the tag and the value is the id.

```
<HwThermAlarmEnum type="XEnumDef">
 <NoAlarm v="0"/>
 <T7300CommFailed v="2"/>
 <AlarmNotifyDisabled v="255"/>
 </HwThermAlarmEnum>
```

A TypeDef contains a set of data elements. Each data element contains a name and set of qualifiers. The "qual" attribute contains a type field(u8, s8, b8, e8 ..), type restrictions (min,max) and encoding (resolution, byteOffset, bitOffset, len) information. For a description of valid element values see Appendix B. If an element is an enumeration then the enumDef attribute must be included to specify the name of the EnumDef used.

```
<HwThermAlarm type="XTypeDef">
 < elem n="subnet" qual="u8 res=1.0 off=0.0"/>
 < elem n="type" qual="e8" enumDef="HwThermAlarmEnum"/>
</HwThermAlarm>
```

A TypeDef element may also include "default" and "engUnit" attributes.

```
<HwThermConfig type="XTypeDef">
 <TODOffset qual="u8 byt=0 bit=0 len=4 min=0.0 max=15.0 "
    default="0" engUnit="F"/>
 <DeadBand qual="ub byt=0 bit=4 len=4 min=2.0 max=10.0 "
    default="2" engUnit="F"/>
</HwThermConfig>
```

A TypeDef may have nonstandard features which require a software implementation. This is the case for typedefs with unions. Unions are not currently supported. A `typeSpec` attribute can be used to specify a class file in a baja module as the implementation of the TypeDef. The class must be a subclass of `BLonData` and provide overrides to `byte[] toNetBytes()` and `fromNetBytes(byte[] netBytes)`.

```
<FileStatus type="XTypeDef">
 < typeSpec v="lonworks:LonFileStatus"/>
</FileStatus>
```

LonDevice

A LonDevice consists of a defined element deviceData and sets of 0 or more of each XNetworkVariable, XNetworkConfig, XConfigProperty, and XMessageTag type elements.

```
<T7300h type="XLonDevice">
 <deviceData> . . . </deviceData>
 <nviRequest type="XNetworkVariable"> . . . </nviRequest>
 <nvoAlarmLog type="XNetworkVariable"> . . . </nvoAlarmLog>
  <nciApplVer type="XNetworkConfig"> . . . </nciApplVe>
 <ScheduleFile type="XConfigProperty"> . . . </ScheduleFile>
 <fx_explicit_tag type="XMessageTag"> . . . </fx_explicit_tag>
</T7300h>
```

DeviceData

DeviceData is a defined set of values need to describe or qualify a lonworks device. A complete list of elements and their default values provided later.

```
<deviceData>
 <majorVersion v="4"/>
```

```
 <programID v="80 0 c 50 3c 3 4 17"/>
 <addressTableEntries v="15"/>
 . . .
</deviceData>
```

NVs, NCIs, ConfigProps

NetworkVariable, NetworkConfig (nci), and ConfigProperty elements share a common structure. Each one consists of a set of defined elements and a data definition. See Appendix A for a complete list of defined elements and their default values.

The data definition can take one of three forms:

1. for standard types a snvtType(for nv/nci) or scptType(for nci/cp) element
2. a typeDef element to specify the XTypeDef containing the data elements
3. a set of elem entries contained in the nv,nci,cp with the same definition as used for TypeDef

```
<nvoAlarmLog type="XNetworkVariable>
 <index v="38"/>
 <direction v="output"/>
 <typeDef="HwThermAlarmLog"/>
</nvoAlarmLog>

<nviRequest type="XNetworkVariable">
 <index v="0"/>
 <snvtType v="objRequest"/>
 . . .
</nviRequest>

<nciSetpoints type="XNetworkConfig">
 <index v="17"/>
 <snvtType v="tempSetpt"/>
 . . .
</nciSetpoints>

<bypassTime type="XConfigProperty">
  <scptType v="CpBypassTime"/>
 <scope v="object"/>
 <select v="0"/>
 . . .
  </bypassTime>
```

File Attribute

There will be cases where it is desirable to nest interface files. This will provide a means to share type definitions between multiple device interface files. It may also ease the process of auto generating the files when the data is contained in multiple forms (i.e. xif files, resource files, ...).

To include a file an element with the "file" attribute is included in the root. The path in the file attribute entry is specified relative to the containing file.

The following is an example of nested files. File #1 contains enum definitions, File #2 contains type definitions which use the enumDefs and file #3 contains the device definition which may use both.

```
File #1  ..\honeywell\enum\ HwThermEnum.xml
<?xml version="1.0" encoding="UTF-8"?>
```

```
<HwThermEnum type="XLonXMLInterfaceFile">
 <HwThermAlarmEnum type="XEnumDef">
  <NoAlarm v="0"/>
  <InvalidSetPtAlrm v="1"/>
  . . .
 </HwThermAlarmEnum>
 . . .
</HwThermEnum>

File #2  ..\honeywell\datatypes\ HwTherm.xml
<?xml version="1.0" encoding="UTF-8"?>
<HwTherm type="XLonXMLInterfaceFile">
 <HwThermEnum  file="..\enum\HwThermEnum.xml"/>
 <HwThermAlarm  type="XTypeDef">
  <elem n="subnet" qual="us res=1.0 off=0.0 "/>
  <elem n="node" qual="us res=1.0 off=0.0 "/>
  <elem n="alarmType" qual="en " enumDef="HwThermAlarmEnum"/>
 </HwThermAlarm>
 . . .
</HwTherm>

File #3  ..\honeywell\
<?xml version="1.0" encoding="UTF-8"?>
<T7300h type="XLonXMLInterfaceFile">
 <HwTherm  file="..\datatypes\HwTherm.xml"/>
 <T7300h type="XLonDevice">
   <nvoAlarm type="XNetworkVariable" >
   <typeDef="HwThermAlarm"/>
   <index v="36"/>
   <direction v="output"/>
...
  </nvoAlarm>
 </T7300h>
</T7300h>
```

XDeviceData Definition

XDeviceData definition: see LonMark External Interface File Reference Guide 4.0B

Type	Name	Default	Valid Values
int	majorVersion	0	-
int	minorVersion	0	-
byte[]	programID	0 0 0 0 0 0 0 0	-
int	domains	2	-
int	addressTableEntries	0	-
boolean	handlesIncomingExplicitMessages	false	-
int	numNvDeclarations	0	-
int	numExplicitMessageTags	0	-
int	networkInputBuffers	0	-

int	networkOutputBuffers	0	-
int	priorityNetworkOutputBuffers	0	-
int	priorityApplicationOutputBuffers	0	-
int	applicationOutputBuffers	0	-
int	applicationInputBuffers	0	-
int	sizeNetworkInputBuffer	0	-
int	sizeNetworkOutputBuffer	0	-
int	sizeAppOutputBuffer	0	-
int	sizeAppInputBuffer	0	-
String	applicationType	unknown	unknown,mip,neuron,hostSelect,hostNISelect
int	numNetworkVariablesNISelect	0	-
int	rcvTransactionBuffers	0	-
int	aliasCount	0	-
boolean	bindingII	false	-
boolean	allowStatRelativeAddressing	false	-
int	maxSizeWrite	11	-
int	maxNumNvSupported	0	-
int	neuronChipType	0	-
int	clockRate	0	-
int	firmwareRevision	0	-
int	rcvTransactionBlockSize	0	-
int	transControlBlockSize	0	-
int	neuronFreeRam	0	-
int	domainTableEntrySize	0	-
int	addressTableEntrySize	0	-
int	nvConfigTableEntrySize	0	-
int	domainToUserSize	0	-
int	nvAliasTableEntrySize	0	-
boolean	standardTransceiverTypeUsed	true	-
int	standardTransceiverTypeId	0	-
int	transceiverType	0	-
int	transceiverInterfaceRate	0	-
int	numPrioritySlots	0	-
int	minimumClockRate	0	-
int	averagePacketSize	0	-

int	oscillatorAccuracy	0	-
int	oscillatorWakeupTime	0	-
int	channelBitRate	0	-
boolean	specialBitRate	false	-
boolean	specialPreambleControl	false	-
String	specialWakeupDirection	input	input,output
boolean	overridesGenPurposeData	false	-
int	generalPurposeData1	0	-
int	generalPurposeData2	0	-
int	generalPurposeData3	0	-
int	generalPurposeData4	0	-
int	generalPurposeData5	0	-
int	generalPurposeData6	0	-
int	generalPurposeData7	0	-
int	rcvStartDelay	0	-
int	rcvEndDelay	0	-
int	indeterminateTime	0	-
int	minInterpacketTime	0	-
int	preambleLength	0	-
int	turnaroundTime	0	-
int	missedPreambleTime	0	-
int	packetQualificationTime	0	-
boolean	rawDataOverrides	false	-
int	rawDataClockRate	0	-
int	rawData1	0	-
int	rawData2	0	-
int	rawData3	0	-
int	rawData4	0	-
int	rawData5	0	-
String	nodeSelfID	""	-

NetworkVariable and Network Config common elements

Type	Name	Default	Valid Values
String	snvtType	"xxx"	from SNVT Master List."SNVT_angle_vel" becomes "angleVel".
int	index	-1	-
int	averateRate	0	-

int	maximumRate	0	-
int	arraySize	1	-
boolean	offline	false	-
boolean	bindable	true	-
String	direction	"input"	input,output
String	serviceType	"unacked"	acked, repeat, unacked, unackedRpt
boolean	serviceTypeConfigurable	true	-
boolean	authenticated	false	-
boolean	authenticatedConfigurable	true	-
boolean	priority	false	-
boolean	priorityConfigurable	true	-

NetworkVariable only elements

Type	Name	Default	Valid Values
String	objectIndex	""	-
int	memberIndex	-1	-
int	memberArraySize	1	-
boolean	mfgMember	false	-
boolean	changeType	false	-

NetworkConfig only elements

Type	Name	Default	Valid Values
String	scptType	""	-
String	scope	"node"	node,object,nv
String	select	""	If for node select=-1. Possible formates are n n~m n-m n.m n/m
String	modifyFlag	"anytime"	anytime, mfgOnly, reset, constant, offline, objDisable, deviceSpecific
float	max	Float.NaN	-
float	min	Float.NaN	-
boolean	changeType	false	-

ConfigParameter elements

Type	Name	Default	Valid Values
String	scptType	""	-
String	scope	"node"	node,object,nv
String	select	""	If for node select=-1. Possible formates are n n~m n-m n.m n/m
String	modifyFlag	"anytime"	anytime, mfgOnly, reset, constant, offline, objDisable, deviceSpecific
int	length	0	-
int	dimension	1	-

float	max	Float.NaN	-
float	min	Float.NaN	-
String	principalNv	""	if the scope is object and the scpt is inherited then this specifies the memberNumber of the principalNv in the selected object. Prefixed with '#' if mfgDefined member '|' if standard member.

Element Qualifier

The format for an element attribute is:

```
qual="Type [qualifier=xx]"
example: qual="u8 res=0.1 min=5 max=12"
```

Type	Device Data Type	Valid Qualifiers
c8	character - 1 byte	-
s8	signed short - 1 byte	res, off, min, max, invld
u8	unsigned short - 1 byte	res, off, min, max, invld
s16	signed long - 2 byte	res, off, min, max, invld
u16	unsigned lon - 2 byte	res, off, min, max, invld
f32	float - 4 byte	res, off, min, max, invld
s32	signed int - 4 bytes	res, off, min, max, invld
b8	boolean - 1 byte	-
e8	enumeration - 1 byte	-
bb	boolean in bit field	byt, bit, len
eb	enumeration in bit field	byt, bit, len
ub	unsigned int in bit field	byt, bit, len, min, max, invld
sb	signed int in bit field	byt, bit, len, min, max, invld
st	string	len
na	no type - byte array	len

Qualifier Code	Description	Default
res	resolution float	1.0
off	Offset	0.0
min	Minimum legal value	Not specified
max	Maximum legal value	Not specified
invld	Invalid value	Not specified
byt	Byte offset - 0 based	-1
bit	Bit offset - 0 based, 7 for msb, 0 for lsb	0
len	Number of bytes(na), char(st) or bits(bb,eb,ub)	1

Build

Contents

This document is structured into the following sections:

- **Overview**
- **Directory Structure**
- **build.xml**
- **module-include.xml**
- **module.palette**
- **module.lexicon**
- **Grouping Modules**
- **devkit.properties**
- **Using Build**

Overview

The Niagara developers kit includes the build tool used by Tridium to build the framework itself. You may utilize this build tool to manage your own Niagara modules. The build tool includes the following:

- The build.exe executable used to invoke the Tridium build software;
- The IBM Jikes compiler is used to compile the java source files;
- Every module declares a "build.xml" file to instruct the build tool how to build and package the module;
- A standard directory structure is required for each module;

Directory Structure

Every module is managed in its own directory structure. Below is an example of a directory for the alarm module:

```
[alarm]
  +- build.xml
  +- module-include.xml
  +- module.palette
  +- module.lexicon
  +- [src]
  |    +- [javax]
  |    |    +- [baja]
  |    |         +- [alarm]
  |    |              +- JavaClass1.java
  |    |              +- JavaClass2.java
  |    +- [com]
  |    |    +- [tridium]
  |    |         +- [alarm]
  |    |              + JavaClass3.java
  |    |              +- [ui]
  |    |                   + BAlarmConsole.java
  |    +- [doc]
  |         +- AlarmConsole-guide.html
  +- [libJar]
```

All of the source code which is used to build the module's jar file is located under the "src" directory. During the build process the "libJar" directory is used to build up the image which will be zipped up for the module's jar file.

build.xml

At the root of every module's directory must be a "build.xml" file. This file instructs the build tool how to build the module. An example of alarm's "build.xml" file:

```
<module
 name = "alarm"
 bajaVersion = "0"
 preferredSymbol = "a"
 description = "Niagara Alarm Module"
 vendor = "Tridium"
>

  <dependency name="baja"  bajaVersion="1" />
  <dependency name="bajaui" />
  <dependency name="workbench" vendor="Tridium" vendorVersion="3.1" />
  <dependency name="history"  />

  <package name="javax.baja.alarm" doc="true" />
  <package name="com.tridium.alarm" />
  <package name="com.tridium.alarm.ui" doc="bajaonly" install="ui" />


  <resources name="com/tridium/alarm/ui/icons/*.png" install="ui" />
  <resources name="doc/*.*" />
</module>
```

The root `module` element must have the following attributes:

- **name** [required]: Name of the module which should match the directory name.
- **bajaVersion** [required]: Baja specification version number, or "0" if not a public specification.
- **preferredSymbol** [required]: Pick a short symbol to use as a abbreviation for the module name. This symbol is used during XML serialization.
- **description** [required]: A short description of the module's purpose in life.
- **vendor** [required]: Vendor for the module.
- **install** [optional]: Sets the default value of install for packages and resources. This attribute is used during provisioning to strip optional directories for headless devices. Valid values are "runtime", "ui", and "doc". The default is "runtime".
- **edition** [optional]: Sets the default value of edition for packages. This attribute determines which Java library edition to compile against. Valid values are "j2me", "j2se", and "j2se-5.0". The default is "j2me".

Note that vendorVersion is not configured in "build.xml", rather it is defined in "devkit.properties"

.

Dependency Element

Use the `dependency` element to declare a module dependency. The dependency is required at both compile time and runtime. There are four possible attributes:

- **name** [required]: Name of dependent module.
- **bajaVersion** [optional]: Lowest Baja specification version required for the module.
- **vendor** [optional]: Specifies a required vendor name for the dependent module.
- **vendorVersion** [optional]: Specifies a required vendor version for the dependent module.

Package Element

The `package` element is used to declare a Java package. These elements are used for both compiling java source files as well as generating reference documentation. The following attributes are supported:

- **name** [required]: Name of package as it would be declared in a `package` or `import` statement. There must be a

corresponding directory "src/packagePath" in your modules directory structure.

- **doc** [optional]: If declared this attribute must be either `false`, `true`, or `bajaonly`. The default is `false`. If `false` then no reference documentation is generated for the package. If `bajaonly` is specified then only `BObject` Types and slot bajadoc is generated. If `true` is specified then full Javadoc documentation is generated on all public Java classes and members. For more information see Deploying Help.
- **compile** [optional]: If declared this attribute must be either `true` or `false`. The default is `true`. This allows you to declare a package for documentation, but not compile the java source code into class files during a build.
- **install** [optional]: This attribute is used during provisioning to strip optional directories for headless devices. Valid values are "runtime", "ui", and "doc". The default is defined by module element.
- **edition** [optional]: This attribute determines which Java library edition to compile against.Valid values are "j2me", "j2se", and "j2se-5.0". The default is "j2me".

Resources Element

The `resources` element is used to copy files from the "src" directories into the "libJar" directories. The following are the attributes supported:

- **name** [required]: Name of files to copy from the "src" to "libJar" directory. The path is relative to the "src" directory and is copied to a matching directory under "libJar". You may specify an explicit filename or use wildcards such as "*.*" or "*.png".
- **install** [optional]: This attribute is used during provisioning to strip optional directories for headless devices. Valid values are "runtime", "ui", and "doc". The default is defined by module element.

module-include.xml

The "module-include.xml" file is an optional file that is placed directly under the module's root directory with the "build.xml". If it is declared, then the build tool automatically includes it in the module's manifest file "meta-inf/module.xml" file. It's primary purpose is to allow developers to declare `def` and `type` elements. An example is provided:

```
<!-- module-include file -->
<defs>
  <def name="alarm.foo" value="something" />
</defs>

<types>
  <type name="AlarmRecord" class="javax.baja.alarm.BAlarmRecord" />
  <type name="AlarmRecipient" class="javax.baja.alarm.BAlarmRecipient" />
  <type name="AlarmClass" class="javax.baja.alarm.BAlarmClass"/>
</types>
```

module.palette

The "module.palette" file is an optional file that is placed directly under the module's root directory. If included it is automatically put into the root of the "libJar" directory, and accessible in the module as "/module.palette". The "module.palette" file should contain the standard palette of public components provided by the module. The format of the file is the same as a standard .bog file.

module.lexicon

The "module.lexicon" file is an optional file that is placed directly under the module's root directory. If included it is automatically put into the root of the "libJar" directory. The lexicon file defines the name/value pairs accessed via the Lexicon API.

Grouping Modules

The Niagara build tool allows you to create hierarchies of modules which may be managed together. This is done through the use of grouping "build.xml" files. Let's look at an example:

```
<module
 name = "drivers"
 specVersion = "1"
```

```
 preferredSymbol = "drivers"
 description = "Niagara Framework Build Driver"
 vendor = "Tridium"
>

  <submodule name="lonworks" />
  <submodule name="bacnet" />
  <submodule name="modbus" />

</module>
```

You will note that it appears very similar to a standard "build.xml" file, except that it only contains `submodule` elements. The `submodule` elements contain a single attribute `name` which maps to the sub-module's root directory. If the modules have inter-dependencies, then they should be listed in order of the dependencies. You may use grouping "build.xml" files to create groupings of groupings and built up projects of arbitary size and depth.

devkit.properties

The build tool uses the "*home*/lib/devkit.properties" file to store a couple key pieces of information:

```
# This property defines the master build number used for vendorVersion.
build=3.0.59

# This property defines the project's root directory where
# the the top level "build.xml" is located.
project.home=d:/niagara/r3dev
```

Using Build

Once you have your directories and "build.xml" files all squared away you use "build.exe" to actually build your modules. The first argument to most build commands is the path of the module or group of modules to build. The second argument is typically the build command to execute, the default is `jar`.

The following command compiles and jars the module "foo" located under the project root (declared in devkit.properties):

```
build /foo
```

The following command cleans the "libJar" directory of the module contained in a directory "goop" relative to the current working directory:

```
build goop clean
```

The following command cleans all modules in the project, then rebuilds them:

```
build / cleanjar
```

Execute the following command to dump a full listing of commands and options supported by the build tool:

```
build -help
```

Deploying Help

Overview

Help documentation is deployed as a set of files zipped up in module jar files. Help content can be any MIME typed file. The primary content types are:

- **HTML**: Niagara provides support for HTML 3.2 with cascading style sheets. See HTML Support in the User Guide for specific tags and attributes supported. This is the main format used to distribute help content.
- **Bajadoc**: The Bajadoc format is a special XML file used to distribute reference documentation. Niagara supports a special plugin, the BajadocViewer which allows users to view the reference documentation in a variety of ways. Bajadoc files are generated from Javadoc comments using the build tool.

In addition to help content, the Niagara Framework also supports delivering navigation data using JavaHelp files. Niagara help system is loosely compliant with version 1.0 of JavaHelp specification from Sun Microsystems. By the term "loosely"; we mean that it is compliant with some of its requirements as of version 1.0.

There are three steps in help content creation:

1. *Developer supplies help content files and help structure files*. Most of help content will be in form of HTML files, maybe with some graphics to enhance the presentation. As a general rule, you as developer should not concern yourself with anything but the content itself, providing HTML files with defined title and body that contains only content-related information. Developers should also include guide help for all their `BPlugins` designed for use by `BComponents`. This documentation is standard HTML files located in the "doc" directory using a naming convention of "*`module-TypeName`*`.html`".
   You should provide a TOC file, to specify the logical order of the help files.
2. *(Optional) Developer supplies lexicon key to point to module containing help*. Guide help (Guide on Target) will look for the HTML file defined above in the doc directory of its module if the `help.guide.base` is not defined in its lexicon. You can supply this key to point to another module. As an example, most core modules point to docUser:

   ```
   help.guide.base=module://docUser/doc
   ```

3. *Build the module*. The module containing the help content is built using the standard build tool. See Using Build. In addition to the standard build, you should use the htmldoc tool to enhance HTML files, and then use the index tool to build search index files. During this step, the help content is indexed for the full text search purposes. For example, to build docDeveloper, we ran this sequence of commands:

   ```
   build /docs/docDeveloper clean
   build /docs/docDeveloper
   build /docs/docDeveloper htmldoc
   build /docs/docDeveloper index
   ```

Help Views

The same help content can be presented in many different ways. Each way of presenting help content is called *view* in JavaHelp lingo. Three most typical *views* are: Table of Contents, API and Search.

- Table of Contents, a.k.a. TOC, is used for presenting help content in a structured way, in some logical order.
- API is used for presenting bajadoc organized by module.
- Search allows full text search of the help content based on some search criteria.

We have added our own "standard" view - BajaDoc view. This is a way of presenting reference documentation for the module classes.

TOC

As a general rule, you should provide a rough TOC with your help content. This should be an XML file, named `toc.xml`, located in the `doc/` directory. This file is required for a module to appear in the help table of contents. Here's the DTD for this file:

```
<!ELEMENT toc (tocitem*)>
  <!ATTLIST toc version CDATA #FIXED "1.0">
  <!ATTLIST toc xml:lang CDATA #IMPLIED>

<!-- an item -->
<!ELEMENT tocitem (#PCDATA | tocitem)*>
  <!ATTLIST tocitem xml:lang CDATA #IMPLIED>
  <!ATTLIST tocitem text CDATA #IMPLIED>
  <!ATTLIST tocitem image CDATA #IMPLIED>
  <!ATTLIST tocitem target CDATA #IMPLIED>
```

It should have `<toc>` as its root element, and a list of files that you want to include in the final TOC, in the logical order. Although TOC structure can be many levels deep, the most likely case will be a flat list of files.
Each file is included via the `<tocitem>` element, that has two attributes: `text` and `target`. The `text` attribute specified the text of the TOC node as it appears in the TOC tree, the `target` attribute can be either relative URL of the help content file associated with this TOC item (relative to the `doc/` directory). It is important that the help file URL is relative to the `doc/` directory. We require that at least one of these attributes is defined.
You may use `tocitem` elements with only the `text` attribute defined as grouping TOC nodes. If you want to define a TOC node associated with some help content, you must provide the `target`. If you provide the `target` only, the `text` will be generated as the name of the target file, without path and extension.
Here's a sample TOC file:

```
<toc version="1.0">
    <tocitem text="Overview" target="overview.html" />
    <tocitem text="User Guide" target="userGuide.html" />
    <tocitem text="Developer Guide" target="devGuide.html" />
</toc>
```

API

This is a list of modules with bajadoc.

Search

This is a search view to search text.

`bajadoc` Command

Every module should include reference documentation. Reference documentation is built using the Niagara build tool:

```
build [moduleDir] bajadoc
```

The module must already have been built using the `build [moduleDir]` command. The .bajadoc files are compiled from the Javadoc found in the source code and placed in the "libJar" directory. Then the module is re-jarred using the new .bajadoc files. For more information on building BajaDoc, see build.xml in build.html.

`htmldoc` Command

htmldoc tool is invoked using the Niagara build tool:

```
build [moduleDir] htmldoc
```

The module must already have been built using the `build [moduleDir]` command.
This tool enhances HTML files. Every HTML file will be enhanced with the following:

- style sheet link - This is hard-coded and not configurable. If your HTML file already has a `link` element in document's `head` section, the auto-generated link will not be inserted.
- copyright notice - The copyright notice generated is controlled by a property in devkit.properties:

```
htmldoc.copyright=Copyright © 2000-%year% Tridium Inc. All rights reserved.
```

If your document has `<p class="copyright">` tag, empty or not, the auto-generated copyright notice will not be inserted in it.

- navigation links - Based on the information in the TOC, navigation links are inserted in every HTML file that doesn't have `<p class="navbar">` or `<div class="navbar">` tag. The navigation links include three links:
  - *Index* - always points to `index.html`
  - *Prev* - points to the previous file in TOC, or disabled if this is the first file in TOC, or if this file is not listed in TOC.
  - *Next* - points to the next file in TOC, or disables if this is the last file in TOC, or if this file is not listed in TOC.

The entire module is then re-jarred, and the enhanced help content is included in the JAR produced.

`index` Command

Index tool is invoked using the Niagara build tool:

```
build [moduleDir] index
```

The module must already have been built using the `build [moduleDir]` command.
Building search indices out of help content.
If you want the `.bajadoc` documentation to also be indexed, you should run the bajadoc command before running the index command.
During this step, all *visible text* inside the help content files will be broken into word tokens and stored in the binary files documents.dat, postings.dat, worddocs.dat and words.dat. The entire module is then re-jarred, and the enhanced help content is included in the JAR produced.

Slot-o-matic 2000

Overview

The **Slot-o-matic 2000** is a java source code preprocessor which generates java source code for Baja slots based on a predefined comment header block. The generated code is placed in the same source file, and all other code in the original source is not modified in any way.

Usage

Invocation

Slot-o-matic is invoked by the executable `slot.exe` . To get help invoke:

```
D:\>slot -?

usage: slot [-f] [-?] <dir | file ...>
-f force recompile of specified targets
-? provides additional help.

slot compiles Baja object files.

For a file to be compiled, it must have a name
  of the form B[A-Z]*.java.

slot will happily recurse any and all directory arguments.
```

Slot-o-matic will compile any file that meets the following conditions:

1. The file name is of format *B[A-Z]*.java*, e.g. *BObject.java* or *BSystem.java*, but not *Ball.java.*
2. The source code has a comment block delimited by `/*- -*/`.

When Slot-o-matic compiles a file, it reads the comment block and generates new java source code based on the contents of that block. The new source is placed in the file being compiled immediately **after** the Baja comment block. If any errors are found, the contents of the file are not altered in any way. The source file may (and indeed probably must) have any other source code required to implement the class in the source file, as with normal java source. The **only** difference between a normal java source file and one usable by Slot-o-matic is the `/*- -*/` comment block.

Compiling a file is simple:

```
D:\>slot D:\niagara\r3dev\fw\history\javax\baja\history\BHistoryService.java
  Compile BHistoryService.java
Compiled 1 files

D:\>
```

As is a directory:

```
D:\>slot D:\niagara\r3dev\fw\history
  Compile BHistoryService.java
Compiled 1 files

D:\>
```

Slot-o-matic works like `make` in that it will only compile files whose /*- -*/ comment block's content has changed since the last compile. To force recompile, use the `-f` flag on a file or directory:

```
D:\>slot -f D:\niagara\r3dev\fw\history\src\javax\baja\history\
  Compile BBooleanHistory.java
  Compile BFloatHistory.java
  Compile BHistory.java
  Compile BHistoryDevicelet.java
  Compile BHistoryJoin.java
  Compile BHistoryPeriod.java
  Compile BHistoryService.java
  Compile BHistorySync.java
  Compile BStorageType.java
Compiled 9 files

D:\>
```

Slot file format

As stated above, Slot-o-matic will compile only files that meet certain conditions. In addition to having an appropriate file (and by extension class) name, the source code in the file must contain a comment block that describes the slots on the object to be compiled.

Examples

Class Example

This example class would resolve in a file named *BImaginaryObject.java*.

```
  /*-
  class BImaginaryObject
  {
    properties
    {
      imaginaryName: String
        -- The imaginary name for the imaginary object.
        default {[ "imaginaryName" ]}
      size: int
        -- The size of the imaginary object.
        flags { readonly, transient }
        default {[ 0 ]}
    }
    actions
    {
      imagine(arg: BComponent)
        -- Imagine something
        default {[ new BComponent() ]}
      create(): BSystem
        -- Create a new imaginary system.
    }
    topics
    {
      imaginationLost: BImaginaryEvent
        -- Fire an event when the object loses its imagination.
```

```
    }
  }
  -*/
```

There are blocks for each of the major slot types: properties, actions, and topics. None of the blocks needs to be present.

Properties Block

Each property has a name and a data type. Comments are specified via the "--" tag per line of comment. All comments are transferred to the javadoc headers of the generated source code but are of course optional. A default value for all properties **must** be specified. The default block is delineated by `{[ ]}` and may have any sequence of java code inside it. Flags on the property may also optionally be specified. For more information on the available flags, see the Flags bajadoc. Slot-o-matic will generate all `get` and `set` methods for the property.

Actions Block

Each action may have 0 or 1 input (formal) arguments, and may optionally return a value. Actions are commented like properties. The input argument, if present, **must** have a default value as with a property. Slot-o-matic will generate the action invocation code; the implementor of the class must provide a do<actionName> method that provides the action implementation.

Topics Block

Each topic specifies a name and an event type that it sends when fired. Slot-o-matic generates code to fire the event.

Enum Example

This example class would resolve in a file named *BImaginaryEnum.java*.

```
  /*-

  enum BImaginaryEnum
  {
    range
    {
      good,
      bad,
      ugly
    }
  }

  -*/
```

Each member of the enumeration is specified.

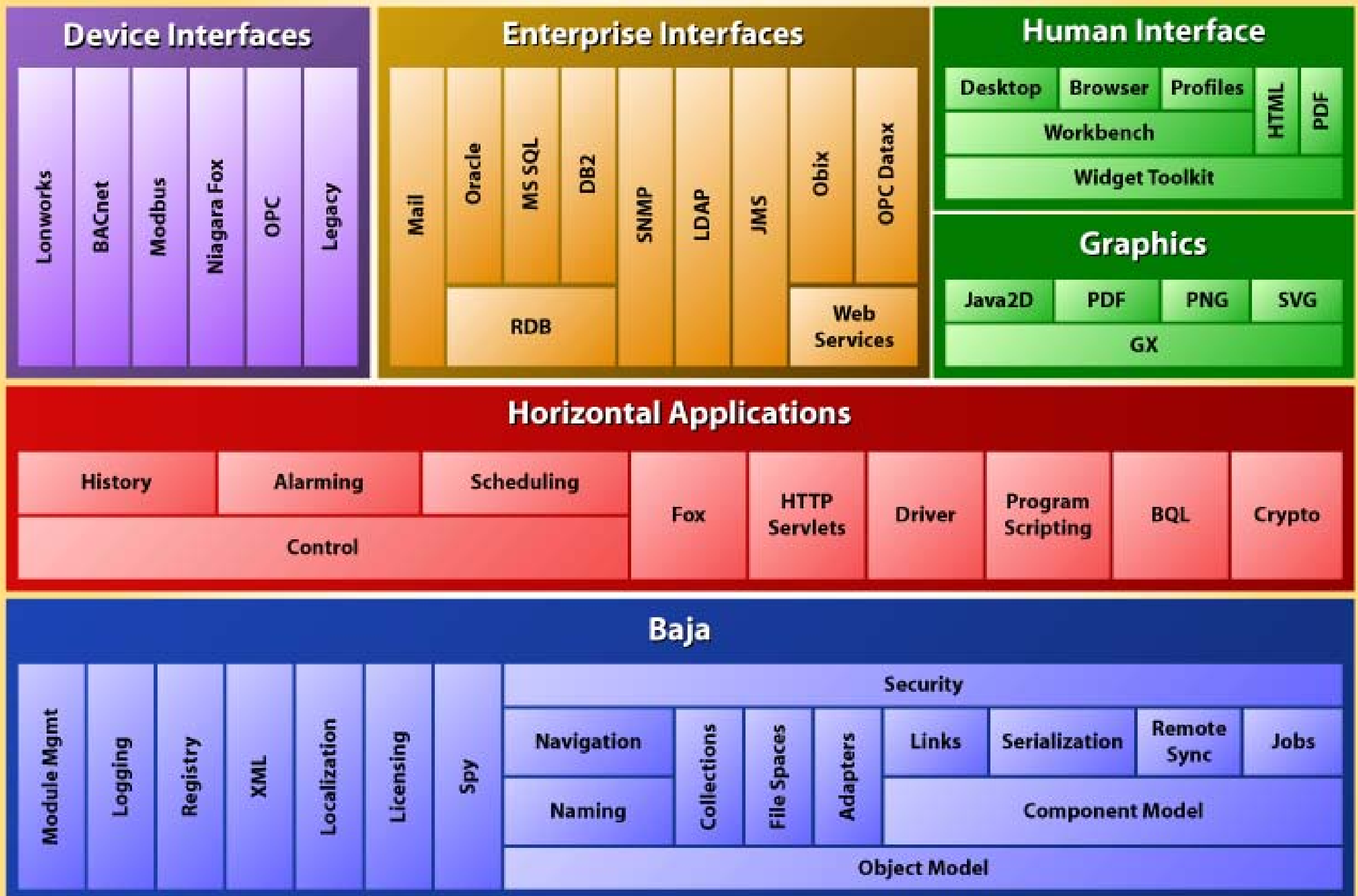
BNF

The formal BNF of the format is as follows:

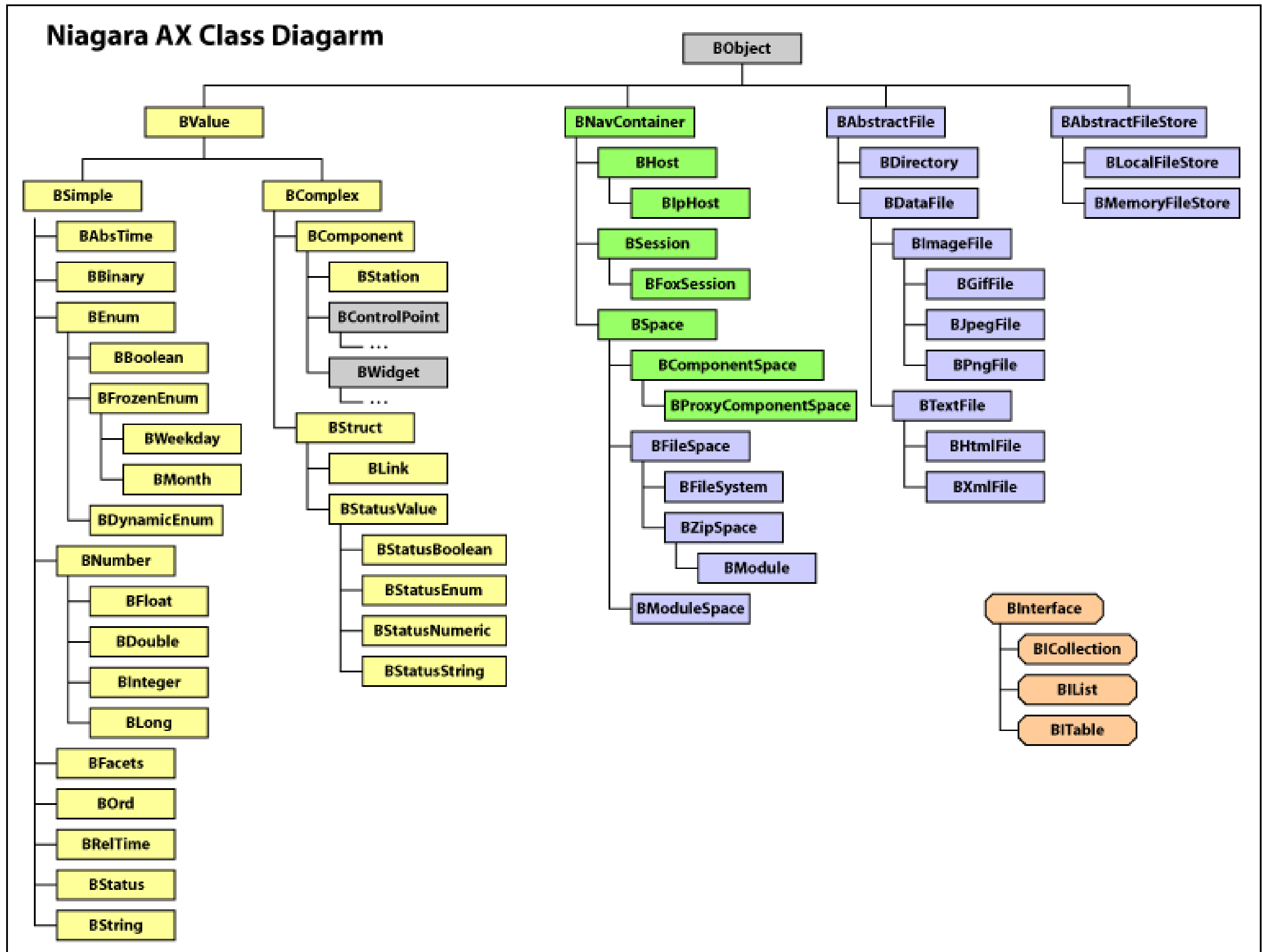
```
<slots definition> = <baja start>
                     <baja header>
                     <slot block>
                     <baja end>

<baja start> ::= /*-
<baja end> ::= -*/

<baja header> ::= <class header> | <interface header> | <enum header>

<class header> ::= class <identifier>
```

```
<interface header> ::= interface <identifier>

<enum header> ::= enum <identifier>

<slot block> ::= "{" <slot def> "}"
<slot def> ::= <slots> | <enum slots>

<slots> ::= { <slotSection> }
<enum slots> ::= <range>

<slotSection> ::= <properties> | <actions> | <events>

<properties> ::= properties <property block>
<property block> ::= "{" { <property> } "}"
<property> ::= <identifer> <type mark> [ <documentation> ] [ <flags> ] <default>

<documentation> ::= <comment>

<flags> ::= flags <flag block>
<flag block> ::= "{" { <flag> { "," <flag> } } "}"
<flag> ::= <readonly | transient | hidden | summary | async | noExecute | defaultOnClone |
userDefined1 | userDefined2 | userDefined3 | userDefined4 }>

<default> ::= default <default block>
<default block> ::= "{" "[" <java expr> "]" "}"

<actions> ::= actions <action block>
<action block> ::= "{" { <action> } "}"
<action> ::= identifier <argument definition> [ <type mark> ] [ <documentation> ]
[ <flags> ] [ <argument default> ]
<argument definition> ::= "(" [ <type decl> ] ")"
<type decl> ::= <identifier> <type mark>
<argument default> ::= <default>

<topics> = topics <topic block>
<topic block> ::= "{" { <topic> } "}"
<topic> ::= <identifier> [ <type mark> ] [ <documentation> ] [ <flags> ]

<range> ::= range <range block>
<range block> ::= "{" <identifier> { "," <identifier> } "}"

<type mark> ::= ":" type
<type> ::= <identifier> { "." <identifier> }

<identifier> ::= <java identifier>

<comment> ::= <single line comment start>
<single line comment> ::= <single line comment start> <comment body>
<single line comment start> ::= "-" "-"
<comment body> ::= { <character> } <new line>
```

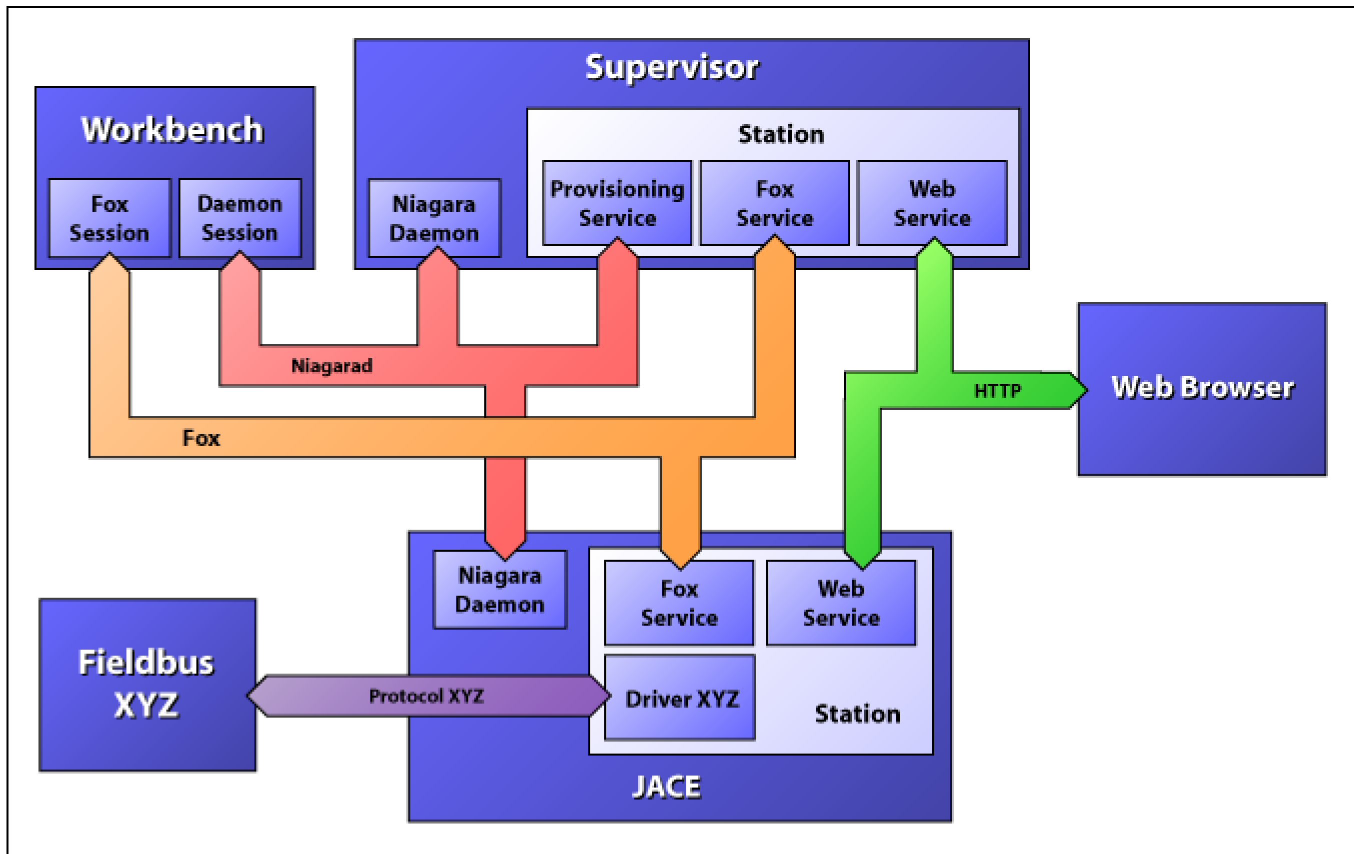
Architecture - Software Stack



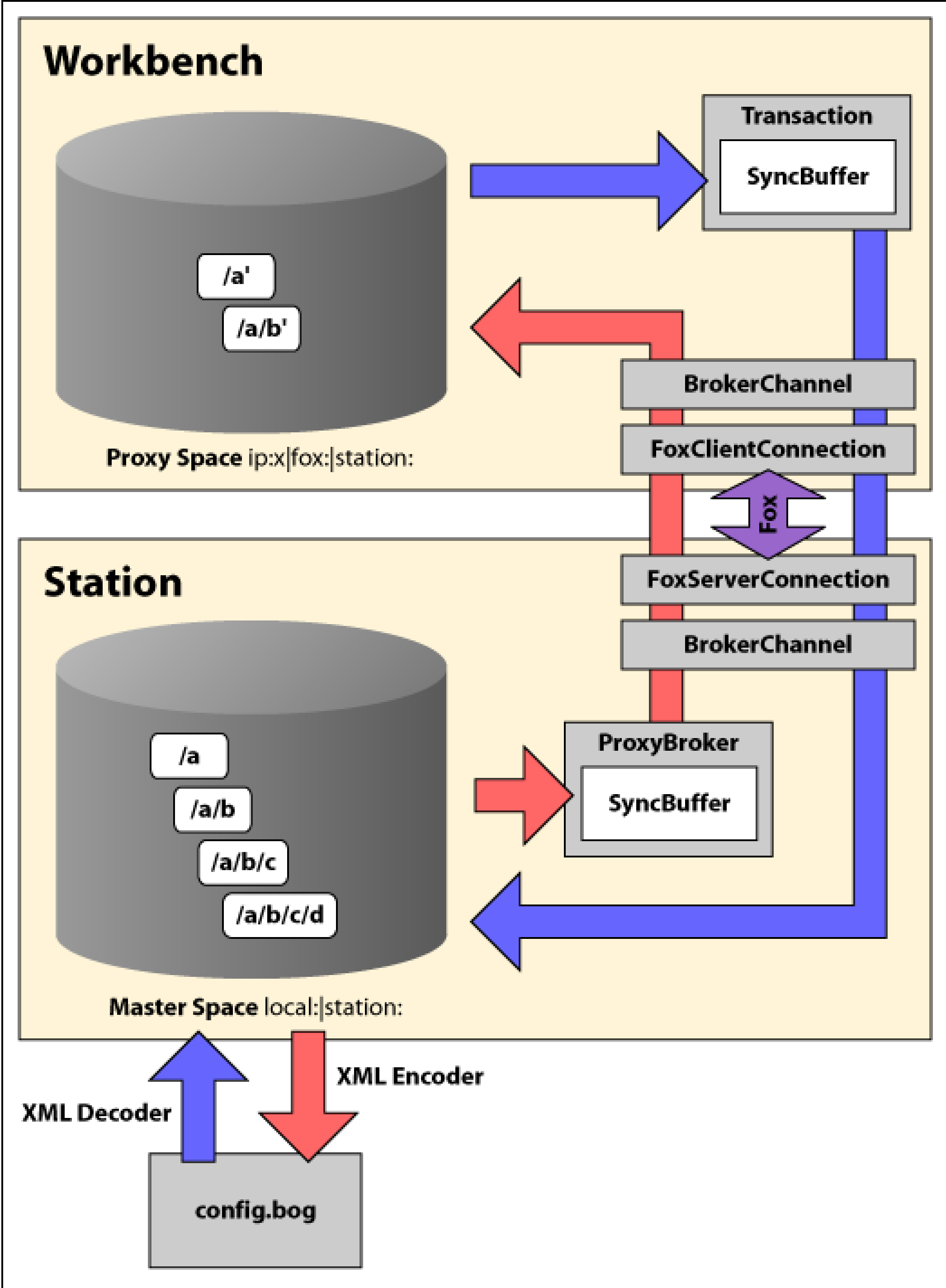
Architecture - Class Diagram



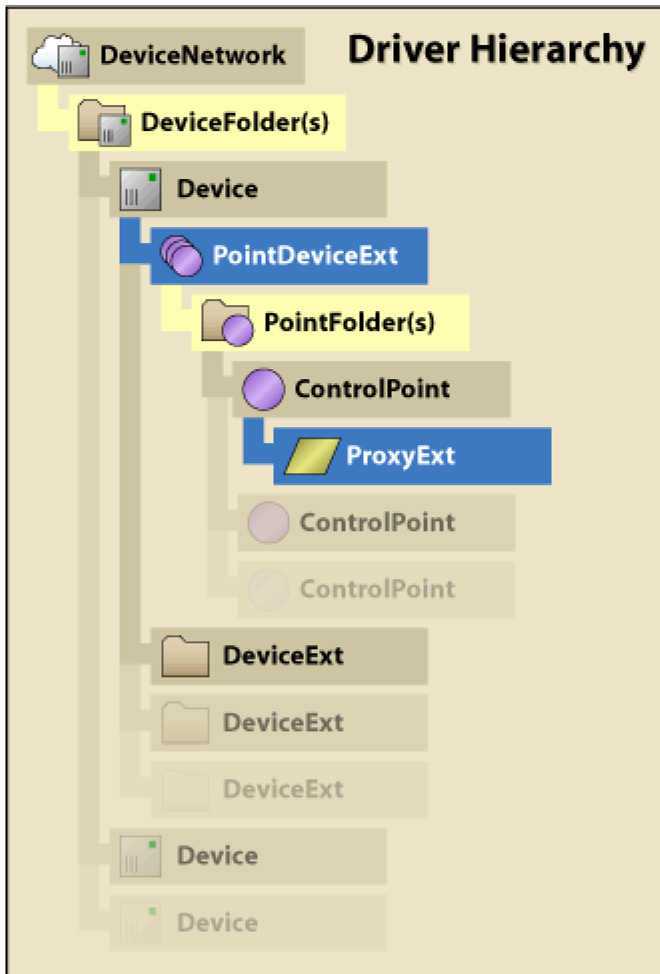
Architecture - Communication



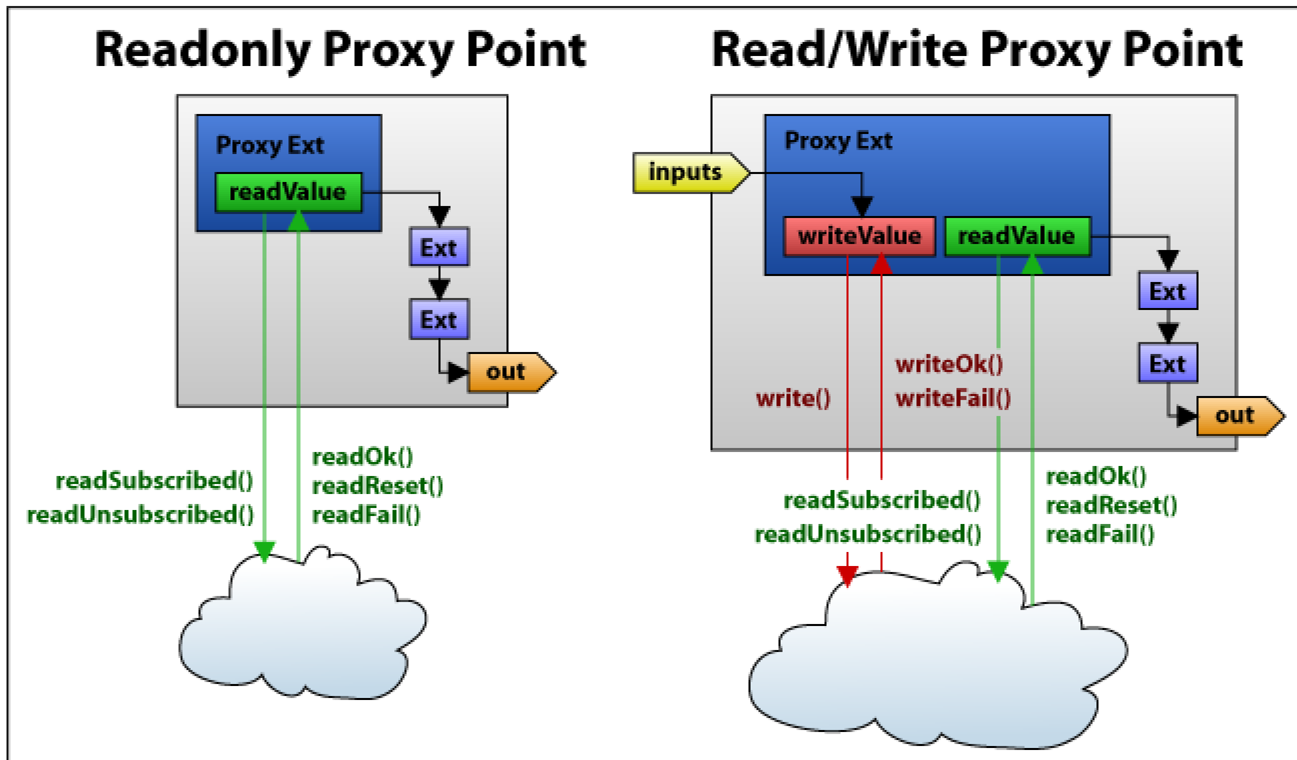
Architecture - Remote Programming



Architecture - Driver Hierarchy



Architecture - ProxyExt



Architecture - Driver Learn