

DYNAMIX CONTEXT PLUG-IN DEVELOPMENT GUIDE

Version 0.7.0

Dr.-Ing. Darren Carlson

1. Introduction

Ambient Dynamix (Dynamix) is an open-source context modeling middleware framework designed to help simplify context-aware software development on resource-constrained mobile devices. Dynamix is both a network-based infrastructure and a device-based framework. The Dynamix Framework runs in the background on a user's device and models contextual information from the environment using a dynamically installed set of Context Plug-ins provisioned from the Dynamix Infrastructure. The resulting stream of contextual information is securely provisioned (or "pushed") to Dynamix-based applications that have registered (and have permission) to receive context events of a specific type. Dynamix applications may also request (or "pull") specific types of context information from the environment on-demand. Dynamix mediates the flow of context information (from plug-ins to applications) using a configurable "Context Firewall", which allows the user of a Dynamix device to precisely manage the type and fidelity of the contextual information provisioned to each application. Dynamix is also capable of installing and updating Context Plug-ins during runtime; allowing a Dynamix device to adapt its context modeling capabilities to better fit the user, the device's operational environment and the availability of new (or updated) plug-ins. To support the creation of a broad range of Context Plug-ins, the Dynamix Project provides an open development process that is designed to encourage contributions by "domain experts" who best understand the subtle and often complex nature of a given context type. By supporting the dynamix integration of externally developed plug-ins and providing a domain-neutral application model, we aim to foster the emergence of a vibrant developer community interested in creating the foundations of next generation of context-aware software.

1.1. Scope

This document describes the Context Plug-ins development process, including the key object models, plug-in lifecycle mechanics, interaction with the Dynamix Framework, deployment packaging and integration testing. This document does not describe the Dynamix Framework architecture in detail nor the Dynamix application development process. If you're interested in contributing to the Dynamix Framework architecture or creating Dynamix-based applications, please see the related documentation and sample code available at <http://ambientdynamix.org>.

1.2. License

Copyright © The Ambient Dynamix Project. All rights reserved.

This documentation and all Ambient Dynamix Project software are licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.3. Table of Contents

1.	Introduction.....	1
1.1.	Scope	1
1.2.	License	1
1.3.	Table of Contents	2
2.	Ambient Dynamix Overview	3
2.1.	Background.....	3
2.2.	Android and OSGi	5
2.3.	Context Plug-in Integration Overview	5
2.4.	Context Plug-in Development Overview	6
2.5.	Source-Code Repository	7
3.	The ContextPluginRuntime.....	7
3.1.	Overview of the Context Plug-in Lifecycle.....	8
3.2.	The ContextPluginRuntime Class.....	8
4.	Implementing the Plug-in Lifecycle	10
4.1.	Android Interaction and Settings Persistence	10
4.2.	Context Plug-in Instantiation.....	11
4.2.1.	The IContextPluginRuntimeFactory Interface	11
4.2.2.	The IContextPluginConfigurationViewFactory Interface	12
4.2.3.	The IContextPluginAcquisitionViewFactory Interface	12
4.3.	Context Plug-in Initialization	14
4.4.	Starting and Stopping a Context Plug-in.....	14
4.5.	Context Plug-in Destruction	14
5.	Context Modeling and Events	14
5.1.	ContextPluginRuntime Types	15
5.1.1.	The PushContextPluginRuntime Class.....	15
5.1.2.	The PullContextPluginRuntime Class.....	16
5.1.3.	The PushPullContextPluginRuntime Class	17
5.2.	Representing Context Information.....	18
5.3.	Sending Context Events	19
5.3.1.	PushContextPluginRuntime Event Utility Methods.....	21
5.3.2.	PullContextPluginRuntime Event Utility Methods	21
5.3.3.	PushPullContextPluginRuntime Event Utility Methods.....	22
5.4.	Power Management	22
6.	Context Plug-in Compilation and Packaging.....	23
7.	Context Plug-in Deployment	25
7.1.	Overview of the Dynamix Infrastructure.....	25
7.2.	Dynamix Infrastructure Interaction.....	26
7.3.	The Dynamix Bootstrap Process.....	26
7.4.	Plug-in Repository Interaction.....	28
8.	Framework Integration and Testing	30
8.1.	Dynamix Client Configuration.....	30
8.2.	Dynamix Client Setup	31
8.3.	Plug-in Integration and Testing	31
9.	Appendix A: The Dynamix Bootstrap Schema	32
10.	Appendix B: The Context Plug-in Description Schema	33

2. Ambient Dynamix Overview

2.1. Background

Mobile devices, such as smart-phones, tablets and net-books, increasingly incorporate powerful microprocessors, feature-rich platform application programming interfaces (APIs) and a rich array of onboard sensors. Using these features, a variety of contextual data can be extracted from the user's environment. Examples of such data include: geo-position strings; orientation primitives; accelerometer values; radio signals; proximate devices or services; light levels; ambient temperature values; camera and microphone data streams; and many more. Applications that can transform these relatively low-level data into meaningful *context information* are often able to adapt more intelligently on the user's behalf. In general, such so-called *context-aware applications* use context information to adapt their functionality to the user's changing situation and operational environment; often optimizing their runtime behaviour or providing entirely new types of functionality.

Transforming low-level environmental data into meaningful context information is often complex and error prone. Issues such as hardware interaction, sensor fusion, noise reduction, protocol handling, time-of-flight calculations, and many others, require significant expertise in a given context domain. Related issues, such as data security and privacy, event caching, power management and the dynamics of mobile code, add to complexity of managing contextual data. Unfortunately, many of these low-level details have very little to do with the actual business logic of context-aware applications, which often simply need a reliable stream of high quality context information and modest mechanisms for acquiring and controlling it. As a result, developers often spend considerable time and expense inventing (or re-inventing) the foundational context modelling mechanisms needed to support their applications.

Dynamix addresses the increasing complexity of context acquisition, modelling and management through a lightweight software framework that provides applications easy access to context domain expertise encapsulated within a set of modular plug-ins that can be installed on-demand. The Dynamix Framework acts as middleware situated between a device's low-level capabilities (e.g. hardware and platform APIs) and a set of Dynamix-based applications, which use Dynamix to request and receive context information. Using Dynamix, applications can register as event listeners, subscribe to context events of specific types, receive context events, request context modelling capabilities, and more.

The actual work of context modelling is performed by a set of dynamically installed Context Plug-ins, which interact directly with a device's underlying capabilities based on a set of permissions (enforced by Dynamix's Plug-in Security Domain). Broadly, a Context Plug-in is an OSGi-based Android 2.x module that utilizes Dynamix's Context Plug-in object model in a structure way (described shortly), which enables the Dynamix Framework to control its lifecycle and allows the plug-in to send context events to registered applications. Context Plug-ins can be downloaded and installed *during runtime* from a variety of network-based repositories. The resulting stream of contextual information provided by Context Plug-ins is securely provisioned (or "pushed") to Dynamix-based applications that have registered (and have permission) to receive context events of a specific type. Dynamix applications may also request (or "pull") specific types of context information from the environment on-demand. Incoming context events provide applications with high-resolution information regarding the situation and environment of the user and/or device. As Dynamix

is domain neutral, applications are free to act upon received context events as needed. An overview of the Dynamix Framework is shown in Figure 1.

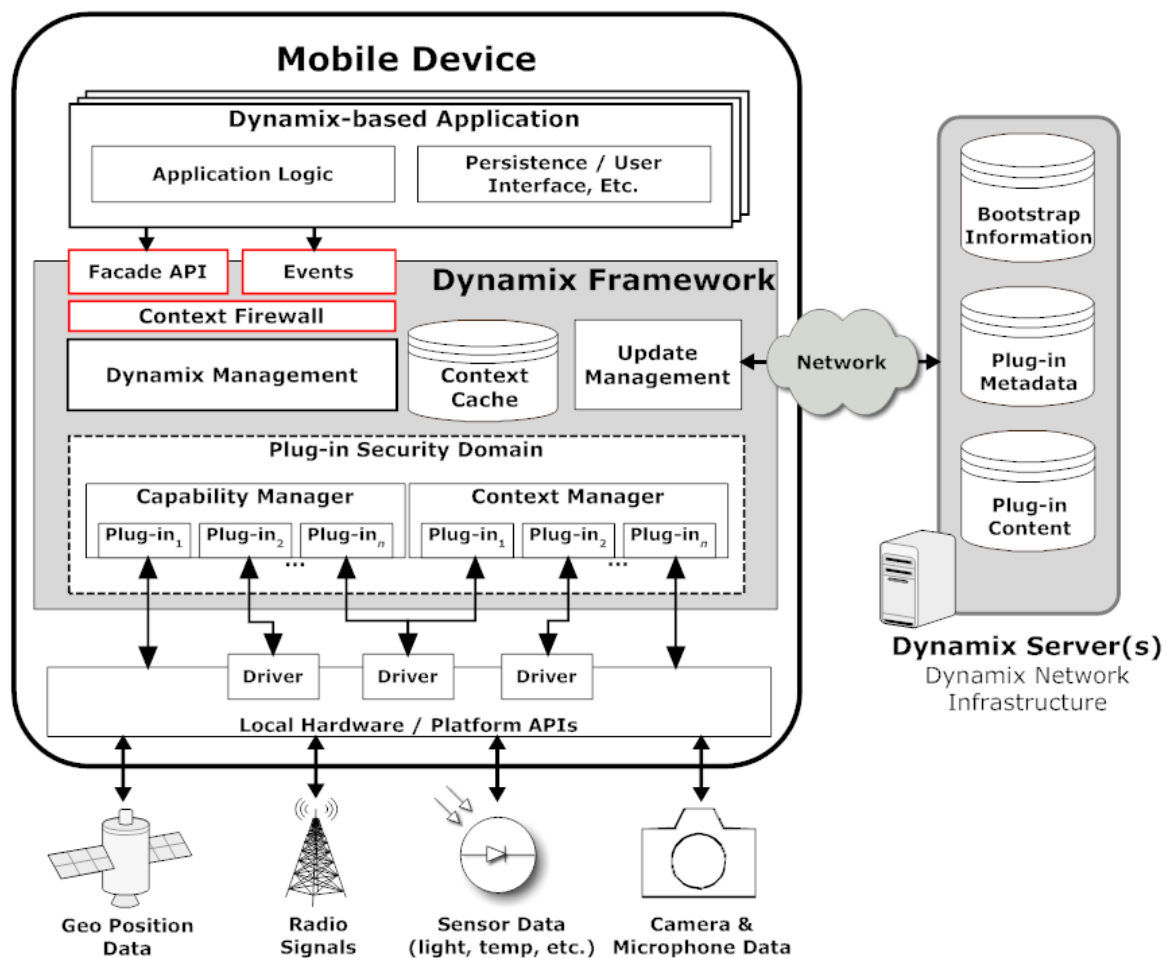


Figure 1: Overview of the Dynamix Framework

Dynamix mediates the flow of context events using a subscription model that filters context information through a “Context Firewall”, which allows the user of a Dynamix device to precisely control the type and fidelity of the contextual information provisioned to each registered application. Dynamix’s Context Firewall prevents applications from accessing critical hardware directly (e.g. camera or microphone data), meaning that registered applications only receive the high-level context information allowed by the user. Unlike the inflexible privacy controls common to many mobile platforms, Dynamix allows users to adjust each application’s privacy settings at any time (not only at installation time), quickly allow and disallow access to particular plug-ins, and precisely adjust the type and fidelity level available to each individual application. The flow of context information can also be suspended globally if needed, which immediately stops all applications from receiving context information.

Dynamix also supports a context cache, which automatically caches the context events generated by Context Plug-ins. The context cache helps applications discover past events and lowers the computational burden of the hosting device by intelligently resending cached events rather than forcing Dynamix to continually re-request (often computationally expensive) context scans.

2.2. Android and OSGi

The Android platform was chosen as the initial deployment target for the Dynamix Framework. As such, Dynamix is designed as an Android Service¹, which runs in the background providing context modeling services to (potentially many) Android applications. The Dynamix Framework operates according to the Android Service Lifecycle and is kept alive as long as there are sufficient device resources (and restarted when possible). Although Dynamix generally serves applications invisibly in the background, users can configure its settings using an inbuilt graphical user interface, which provides a means of controlling the context firewall, managing permissions, performing updates, adjusting plug-in settings, etc. Dynamix also alerts users of important status changes using Android's notification system. Dynamix-based applications are developed using the Android application model², meaning that Dynamix applications are simply standard Android applications that interact with the Dynamix service using Android's ServiceConnection inter-process communication (IPC) approach³. Once the user has authorized the application, it can subscribe to context events and control various aspects of the framework using Dynamix's Façade API and associated Event System, which are described in detail throughout this document.

To manage Context Plug-ins internally, the Dynamix Framework utilizes an industry standard OSGi container⁴, which is used to dynamically integrate mobile code, provide secure access to local resources, impose a comprehensive plug-in life-cycle model (including thread priority), handle dependency management and enforce plug-in versioning. Specifically, Dynamix uses the Apache Felix OSGi container, which is embedded *within* the Dynamix Framework as a private Java module system. Using its embedded OSGi container, Dynamix is capable of downloading, installing (or updating) and integrating Context Plug-in Bundles during runtime without restarting the framework. Context Plug-ins are loaded into the Dynamix Framework's runtime process (with additional security constraints) and operate in accordance with a tightly controlled lifecycle. Context Plug-ins are created as standard OSGi bundles⁵, which are packaged and deployed as Java Archive (JAR) files with additional OSGi metadata.

2.3. Context Plug-in Integration Overview

Context Plug-ins are dynamically integrated into a Dynamix Framework instance as follows:

1. The Dynamix Framework is installed on a mobile device. Once the Dynamix Framework starts (as an Android Service), applications can bind to Dynamix's Façade API and register for context events.
2. Through an update, application request, or as directed by capability analyses, Dynamix discovers a Context Plug-in to install, which is stored in the Dynamix network infrastructure.
3. The OSGi Bundle holding the Context Plug-in's code (and any associated libraries) is downloaded and installed into Dynamix's embedded OSGi framework.
4. After the Context Plug-in is verified, and the requisite permissions are granted by the user, the Context Plug-in is instantiated by the Dynamix Framework using a factory object provided by the Context Plug-in developer.

¹ <http://developer.android.com/reference/android/app/Service.html>

² <http://developer.android.com/guide/topics/fundamentals.html>

³ <http://developer.android.com/reference/android/content/ServiceConnection.html>

⁴ <http://www.osgi.org/About/WhatIsOSGi>

⁵ <http://www.osgi.org/javadoc/r4v42/org/osgi/framework/Bundle.html>

5. After the Context Plug-in is instantiated, Dynamix initializes it, whereby the plug-in acquires all necessary resources and prepares its internal state. During initialization, Dynamix passes the Context Plug-in the current power scheme and (optionally) a settings object, which is used to setup state. (Note: The Dynamix Service provides private settings persistence for each Context Plug-in.)
6. If the Context Plug-in successfully initializes, it may be started at some point. The actual start logic depends on the type of Context Plug-in involved; however, starting typically means that the Context Plug-in begins scanning the environment for context data and modeling context information. (Note: Each Context Plug-in defines a set of native context information types that are used to represent the modeled context information.)
7. As context information is modeled from the environment, Context Plug-ins use Dynamix's predefined event utility methods to send the context events to the Dynamix Framework. Context events are composed of a set of the Context Plug-in's native context information types along with associated fidelity levels, which describe the precision of the context information.
8. The Dynamix Framework caches the incoming context events and then passes the events (and associated fidelity levels) to its Context Firewall, which determines which applications will receive them. Only events that match a specific context event subscription and pass the Context Firewall privacy filters are sent to a registered application. (The user can change the privacy settings of the Context Firewall at any time.)
9. Dynamix continues to operate in this fashion, potentially downloading additional Context Plug-ins, which all operate in parallel using the same mechanisms described above.

2.4. Context Plug-in Development Overview

Context Plug-in development consists of a hierarchical set of tasks, which are designed to structure a Context Plug-in so that it can be dynamically integrated into the Dynamix Framework on-demand. An overview of these tasks and their related document sections are shown in Table 1.

Development Task	Description
1. ContextPluginRuntime Extension See section: 3	Extend one of three base ContextPluginRuntime type classes, which provide a Context Plug-in its basic structural foundation.
2. Implementing the Plug-in Lifecycle See section: 4	Define how Dynamix instantiates, initializes and destroys the Context Plug-in. Optionally: Design configuration user interfaces.
3. Context modeling and Events See section: 5	Implement the IContextInfo interface for each type of context information modeled by the Context Plug-in. Create the context modeling logic matching the chosen ContextPluginRuntime type. Define context events and use Dynamix's utility methods to send them. Optionally: Design context modeling user interfaces.
4. Context Plug-in Compilation and Packaging See section: 6	Compile and package the Context Plug-in as a Dynamix-configured OSGi Bundle.
5. Context Plug-in Deployment See section: 7	Describe the Context Plug-in using Dynamix's configuration schemas. Deploy the Context Plug-in Bundle and its associated description document to a Dynamix plug-in repository.
6. Framework Integration and Testing See section: 8	Configure the Dynamix Framework to integrate the Context Plug-in and perform integration testing.

Table 1: Overview of Context Plug-in Development Tasks and Related Document Sections

2.5. Source-Code Repository

The source-code for developing Context Plug-ins is available from the Dynamix subversion repository located at <http://ambientdynamix.svn.sourceforge.net>. Context Plug-ins are dependent on the “DynamixApplicationApi” and the “DynamixContextPluginApi/” sub-projects, which are collectively called the “Context Plug-in object model”. The Context Plug-in object model consists of the following packages: `org.ambientdynamix.contextplugin.api`; `org.ambientdynamix.contextplugin.api.security`; and `org.ambientdynamix.application.api`.

3. The ContextPluginRuntime

The foundation of Context Plug-in development is the ContextPluginRuntime, which provides a basic set of lifecycle methods that Context Plug-in implementations must extend in order to integrate with the Dynamix Framework. The ContextPluginRuntime actually exists as a hierarchy of classes. The base of the hierarchy is the abstract ContextPluginRuntime class, which provides several lifecycle methods (e.g. `init`, `start`, `stop`, `destroy`, etc.) designed to enable Dynamix to control the Context Plug-in during runtime. Context Plug-in implementations do not extend the abstract ContextPluginRuntime class directly, however. Rather, they extend a particular subclass of the ContextPluginRuntime, depending on their type. Currently, there are three different subclasses (types) of ContextPluginRuntime, each with different context modeling logic and utilities for sending events to the Dynamix Framework (see section 5). The three types are: `PushContextPluginRuntime`, `PullContextPluginRuntime` and `PushPullContextPluginRuntime`. Context Plug-in developers are free to choose the ContextPluginRuntime type that best suits their particular context domain. A simplified version of the ContextPluginRuntime class hierarchy is shown in Figure 2.

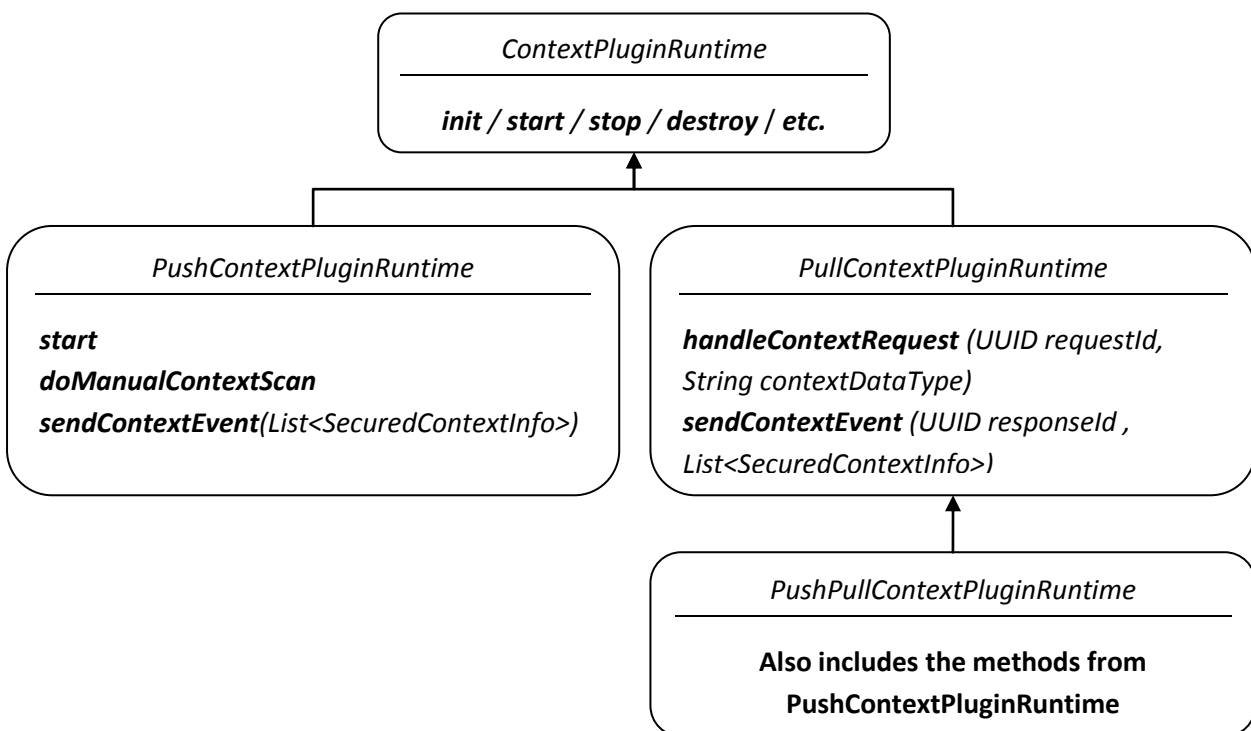


Figure 2: Simplified ContextPluginRuntime Class Hierarchy

3.1. Overview of the Context Plug-in Lifecycle

A ContextPluginRuntime transitions through a series of defined states during its lifecycle, which indicate which methods that can be called on it and provides the Dynamix Framework management information. Plug- states are encapsulated by the PluginState object. The available PluginState types are shown in Table 2.

PluginState: Field Summary	
<code>static PluginState</code>	DESTROYED Plug-in is stopped and ready for garbage collection (all resources have been released).
<code>static PluginState</code>	ERROR Plug-in has entered an error state and is NOT running (all resources have been released).
<code>static PluginState</code>	INITIALIZED Plug-in is initialized and stopped (all necessary resources have been acquired).
<code>static PluginState</code>	NEW Plug-in is new and uninitialized.
<code>static PluginState</code>	STARTED Plug-in is started and running.

Table 2: Overview of PluginState Fields

With reference to Table 2, the Context Plug-in lifecycle is described as follows:

1. When the Context Plug-in is first instantiated, its state is **NEW**.
2. During runtime, the Dynamix Framework will initialize each plug-in. During initialization, a plug-in prepares up internal state (e.g. settings) and acquires the resources it needs to run. If the plug-in successfully initializes, its state becomes **INITIALIZED**. If not, its state becomes **ERROR**.
3. Once a plug-in is **INITIALIZED** it can be started by the Dynamix Framework. If the plug-in is started, its state becomes **STARTED**. If the plug-in stops, its state reverts back to **INITIALIZED**.
4. At the completion of its lifecycle, a plug-in will be destroyed by the Dynamix Framework. During destruction, a plug-in releases any acquired resources and sets its state to **DESTROYED**.

3.2. The ContextPluginRuntime Class

As shown in Figure 2, the foundation of all Context Plug-ins is the abstract ContextPluginRuntime class, which includes basic lifecycle methods. Although concrete Context Plug-in implementations do not extend ContextPluginRuntime directly, each must override the ContextPluginRuntime's abstract methods, as described in Table 3.

ContextPluginRuntime: Abstract Methods

abstract void	destroy() Stops the runtime (if necessary) and then releases all acquired resources in preparation for garbage collection. Once all resources have been released, the runtime's state should be changed to DESTROYED. Once a runtime has been destroyed, it may not be re-started, and will be reclaimed by garbage collection at some indeterminate time in the future.
abstract boolean	init (PluginPowerScheme powerScheme, ContextPluginSettings settings) Called once when the ContextPluginRuntime is first initialized. The implementing subclass should acquire the resources necessary to run. If initialization is successful, this method should return true and set its PluginState to INITIALIZED. If not, the method should release any acquired resources, clean up, set its PluginState to ERROR and return false. Note that the init method may receive a ContextPluginSettings object that can be used to setup internal state.
abstract void	setPowerScheme (PluginPowerScheme scheme) Sets this runtime's PluginPowerScheme to the incoming value. Note that runtimes must be capable of <i>dynamically</i> adjusting their power consumption.
abstract void	start() Called by the Dynamix Context Manager to start (or prepare to start) context acquisition and modeling. When called, the runtime should immediately change its state to STARTED and prepare to handle any additional methods that may be called during STARTED. Implementations may have additional runtime behavior defined in subclasses, such as those provided by PushContextPluginRuntime and PullContextPluginRuntime. <ul style="list-style-type: none"> • Note 1: Implementations must remain STARTED until 'stop' or 'destroy' are called. • Note 2: This method may only be called when the ContextPluginRuntime is in PluginState.INITIALIZED. • Note 3: Implementations should endeavor to conserve local resources and battery power according to the current PluginPowerScheme (which may change over time). • Note 4: If PluginPowerScheme.MANUAL is set, this method should not perform context scans until 'doManualContextScan' is called. • Note 5: This method will be called on a dedicated thread, so the method may block.
abstract void	stop() Called by the Dynamix Context Manager to stop context acquisition and modeling. When called, the runtime should immediately stop modeling context and change its state to INITIALIZED.
abstract void	updateSettings (ContextPluginSettings settings) Called when new settings are available.

Table 3: Key ContextPluginRuntime Methods

4. Implementing the Plug-in Lifecycle

The following section describes how to implement the foundational lifecycle of a Context Plug-in. The section covers instantiation, initialization, starting and stopping, and destruction. The context modeling logic and event generation provided by started Context Plug-ins is presented in section 5.

4.1. Android Interaction and Settings Persistence

Due to Dynamix security constraints, runtimes *cannot* directly access the local file-system, most hardware interfaces and most Android platform APIs. Rather, runtimes receive an `IAndroidFacade` object during instantiation, which provides a secured mechanism for these purposes. Each time the plug-in is instantiated, its runtime is configured with a globally unique `sessionId`, which is to authenticate the runtime during secure interactions with the Dynamix Framework when making method calls using the `IAndroidFacade`. The runtime's `getSessionId()` method can be used to retrieve this id during runtime. An overview of the methods provided by the `IAndroidFacade` is shown in Table 4.

IAndroidFacade: Method Summary	
<code>ContextPluginSettings</code>	<code>getContextPluginSettings</code> (UUID <code>sessionId</code>) Returns the <code>ContextPluginSettings</code> persisted for the given <code>ContextPluginRuntime</code> in the Dynamix Framework.
<code>SecuredContext</code>	<code>getSecuredContext</code> (UUID <code>sessionId</code>) Returns a secured version of the Android Context that is customized for the caller.
<code>boolean</code>	<code>setPluginConfiguredStatus</code> (UUID <code>sessionId</code> , <code>boolean configured</code>) Sets the <code>ContextPluginRuntime</code> 's associated <code>ContextPlugin</code> to the configured or unconfigured state.
<code>boolean</code>	<code>storeContextPluginSettings</code> (UUID <code>sessionId</code> , <code>ContextPluginSettings settings</code>) Stores the <code>ContextPluginSettings</code> in the Dynamix Framework on behalf of the calling <code>ContextPluginRuntime</code> .

Table 4: IAndroidFacade Methods

The `IAndroidFacade`'s `getSecuredContext` method can be used by runtimes to interact with the Android platform (e.g. to request services, etc.). The return type of this method is a `SecuredContext`, which is a secured version of the conventional Android Context⁶ that is customarily used to interact with the Android platform. `SecuredContext` extends `android.content.Context` and is customized with a set of permissions that are controlled by the user. These permissions allow the user to specify which parts of the Android platform are available to each Context Plug-in. As an example, the `SecuredContext`'s

⁶ <http://developer.android.com/reference/android/content/Context.html>

`getSystemService(String)` method may allow one particular Context Plug-in access to the Android platform's `LOCATION_SERVICE`, but disallow access to all other system services, meaning that a call to retrieve the `POWER_SERVICE` will return null. Other runtime's may be configured differently. **For a detailed overview of the types of Android platform access allowed by a `SecuredContext`, please review the JavaDocs located here.**

Many Context Plug-ins require configuration before they can be used. To support configuration, each runtime is able to store and retrieve a `ContextPluginSettings` object, which is simply a `HashMap` of strings that are stored on behalf of `ContextPlugins` by the Dynamix Framework, and are provided to runtimes during initialization and on-demand via the `IAndroidFacade`'s `getContextPluginSettings` method. If user interaction is required to acquire settings, the runtime's View factory can be set to an Android-based user interface designed to help users capture settings (described in section 4.1). Runtimes can call `storeContextPluginSettings` to store settings at any time, and `setPluginConfiguredStatus` to indicate whether or not they are properly configured. Runtimes that are not configured will not be started by the Dynamix Framework; however, once runtimes set their configured status to true, they will be automatically started, if needed.

4.2. Context Plug-in Instantiation

Because the Dynamix Framework downloads and installs Context Plug-ins during runtime, it provides factory mechanisms that allow plug-in instances (and specific user interfaces) to be dynamically created. Context Plug-in instances are created by Dynamix using an `IContextPluginRuntimeFactory`, which is implemented by the Context Plug-in developer for the specific plug-in type. Due to the security constraints of the Android Platform, which prevents the creation of unregistered Activities (the basis of Android user interfaces), Dynamix also provides additional factory mechanisms that allows Context Plug-ins to inject their user interfaces into a host Activity provided by the Dynamix Framework. The following sections describe each of these factory mechanisms.

4.2.1. The `IContextPluginRuntimeFactory` Interface

`ContextPluginRuntimes` are dynamically created by the Dynamix Framework using an implementation of an `IContextPluginRuntimeFactory`, which is registered by the Dynamix Framework as an OSGi service. The implementing class of this service *must* extend the `ContextPluginRuntimeFactory` object (described shortly). Each Context Plug-in provides its own creational factory. Using this factory, runtimes can be created each time Dynamix starts. The `IContextPluginRuntimeFactory` interface contains a single factory method, as shown below in Table 5.

IContextPluginRuntimeFactory: Method Summary

<code>ContextPluginRuntime</code>	<code>makeContextPluginRuntime</code> (<code>ContextPlugin</code> parentPlugin, <code>IAndroidFacade</code> facade, <code>IPluginEventHandler</code> handler, <code>UUID</code> sessionId) Creates a concrete <code>ContextPluginRuntime</code> using the incoming <code>ContextPlugin</code> as the parent.
-----------------------------------	---

Table 5: The `IContextPluginRuntimeFactory` Factory Method

4.2.2. The IContextPluginConfigurationViewFactory Interface

Since dynamically loaded Context Plug-ins cannot provide their own Activities (due to Android's security restrictions), the IContextPluginConfigurationViewFactory provides a means of programmatically creating an Android View⁷ that provides a graphical user interface for Context Plug-in's configuration. As an example, a Context Plug-in that scans a specific social network for status updates may require security credentials from the user to access their profile. The IContextPluginConfigurationViewFactory, in this case, could provide a View that allows the user to enter his or her security credentials using standard text boxes and buttons. The configuration settings generated by such an interaction can be stored by Dynamix on behalf of the plug-in using the persistence facilities described in section 4.1.

Views generated by the IContextPluginConfigurationViewFactory are injected into a hosting Activity provided by the Dynamix Framework. Currently, Views must be created programmatically and *cannot* utilize Android's XML-based Activity creation techniques. A summary of the IContextPluginConfigurationViewFactory methods is shown in Table 6.

IContextPluginConfigurationViewFactory: Method Summary

void	destroyView() Destroy the Android View.
View	initializeView (Context context, ContextPluginRuntime runtime, int titleBarHeight) Creates and Android View using the incoming Android context ⁸ , runtime and title-bar height.

Table 6: IContextPluginConfigurationViewFactory Method Summary

4.2.3. The IContextPluginAcquisitionViewFactory Interface

Since dynamically loaded Context Plug-ins cannot provide their own Activities (due to Android security restrictions), the IContextPluginAcquisitionViewFactory provides a means of programmatically creating an Android View⁹ that provides a graphical user interface for controlling the context acquisition process of a specific Context Plug-in. As an example, a barcode reader Context Plug-in may need the user to operate the device's camera to control the context modeling process (e.g. by pointing the camera at the bar code). Views generated by the IContextPluginAcquisitionViewFactory are injected into a hosting Activity provided by the Dynamix Framework. Currently, Views must be created programmatically and *cannot* utilize Android's XML-based Activity creation techniques. A summary of the IContextPluginConfigurationViewFactory methods is shown in Table 7.

⁷ <http://developer.android.com/reference/android/view/View.html>

⁸ <http://developer.android.com/reference/android/content/Context.html>

⁹ <http://developer.android.com/reference/android/view/View.html>

IContextPluginConfigurationViewFactory: Method Summary

void	destroyView() Destroy the view.
View	initializeView (Context context, PullContextPluginRuntime runtime, UUID requestId, int titleBarHeight) Initialize the View using the Android context ¹⁰ , runtime, requestId and title-bar height.

Table 7: IContextPluginConfigurationViewFactory Method summary

In order to allow Dynamix to dynamically create a Context Plug-in, and any necessary interfaces, plug-in developers must specify which classes should be created at runtime using a specifically configured version of the ContextPluginRuntimeFactory class. This class implements the IContextPluginRuntimeFactory previously mentioned. The ContextPluginRuntimeFactory is configured by simply specifying the ContextPluginRuntime to be created in its constructor. Optionally, two additional user interface classes may also be specified, which are used for configuration and context acquisition. Configuration of the ContextPluginRuntimeFactory is completed using its constructor, which is shown in Table 8.

ContextPluginRuntimeFactory: Constructor Summary

```
ContextPluginRuntimeFactory(Class<? extends ContextPluginRuntime> runtimeClass,  
Class<? extends IContextPluginAcquisitionViewFactory> acquisitionViewFactory,  
Class<? extends IContextPluginConfigurationViewFactory> settingsViewFactory)
```

Table 8: The ContextPluginRuntime Constructor

Based on the ContextPluginRuntimeFactory constructor shown in Table 8, the sample code below illustrates how a ContextPluginRuntimeFactory can be configured to create a SampleContextPluginRuntime.

```
public class PluginFactory extends ContextPluginRuntimeFactory {  
  
    public PluginFactory() {  
  
        super(SampleContextPluginRuntime.class,  
              null,  
              ConfigurationViewFactory.class);  
  
    }  
  
}
```

In the above code, the PluginFactory specifies SampleContextPluginRuntime as the ContextPluginRuntime Class, no context acquisition factory Class (indicated by null), and ConfigurationViewFactory as the configuration view factory Class.

¹⁰ <http://developer.android.com/reference/android/content/Context.html>

4.3. Context Plug-in Initialization

Once a Context Plug-in's runtime has been dynamically instantiated by the Dynamix Framework, it is then initialized, whereby it sets up internal state and acquires resources necessary for it to run. Context Plug-in initialization occurs when the Dynamix Framework starts, either as directed by a user or automatically (e.g. when the device finishes booting). When Dynamix starts, it issues initialization method calls to each of its installed Context Plug-ins. If initialization is successful, the runtime should return true and set its `PluginState` to `INITIALIZED`. If not, the runtime should release any acquired resources, clean up, set its `PluginState` to `ERROR` and return false. Recall from section 3.2, that the signature of the initialization method is: `init(PluginPowerScheme powerScheme, ContextPluginSettings settings)`.

The first parameter of the initialization method is a `PluginPowerScheme`, which indicates the current power management state of the Dynamix Framework (for details, see section 5.4). The second parameter of the initialization method is a `ContextPluginSettings` object, which can be used by the plug-in to setup state based on stored configuration settings. The runtime will receive the last `ContextPluginSettings` object stored, or null if no settings have been stored for the runtime. For reference, the `ContextPluginSettings` object was described in section 4.1.

4.4. Starting and Stopping a Context Plug-in

After a Context Plug-in has been successfully initialized, it may be started by the Dynamix Framework, whereby the runtime should start (or prepare to start) context acquisition and modeling. When `start()` is called, the runtime should immediately change its state to `STARTED` and prepare to handle any additional methods that may be called during `STARTED`. Note that the actual behavior of this method is dependent on the type of `ContextPluginRuntime` chosen by the developer, as discussed in section 5.1.

Dynamix may also stop runtimes at any point, indicating that the runtime should stop context modeling, but *not* release acquired resources or clean up state. When `stop()` is called, the runtime should immediately stop modeling context, stop sending events and change its state to `INITIALIZED`. Note that a runtime may be started again at any time with another call to `start()`, whereby the runtime should again start (or prepare to start) context acquisition and modeling processes.

4.5. Context Plug-in Destruction

When the Dynamix Framework shuts down, all Context Plug-in runtimes are destroyed. During destruction, each runtime must release any acquired resources, clean up internal state (e.g. save state) and prepare for garbage collection. Dynamix destroys Context Plug-in runtimes by calling the runtime's `destroy()` method. Note that Dynamix may initialize and or destroy Context Plug-ins possibly many times during typical usage.

5. Context Modeling and Events

The primary purpose of a Context Plug-in is to perform context modeling, whereby a Context Plug-in's runtime scans the user's environment for contextual data, transforms these data into meaningful context information, encodes this context information into suitable representation formats, and sends the resulting representations to Dynamix as context events. Context Plug-ins never communicate directly with Dynamix-applications; rather, the Dynamix Framework handles provisioning the incoming context events to registered applications based on the privacy settings of its Context Firewall. The specifics of how context information is extracted from the environment are entirely dependent on the underlying context domain; hence, the

process of context modeling can be implemented in various ways by different Context Plug-ins. Recall from section 3, that the foundation of all Context Plug-in's is the `ContextPluginRuntime`, which provides a basic set of lifecycle methods that Context Plug-in implementations must extend in order to integrate with the Dynamix Framework. Context Plug-in implementations do not extend the abstract `ContextPluginRuntime` class directly, however. Instead, they extend a particular subclass of the `ContextPluginRuntime`, depending on their type.

5.1. ContextPluginRuntime Types

To support a wide variety of context domains, Dynamix provides several subclasses (types) of the `ContextPluginRuntime` class, each with a different context modeling logic and associated event sending utility methods. Currently, there are three different subclasses (types) of `ContextPluginRuntime`, each with different context modeling logic and utilities for sending events to the Dynamix Framework. Context Plug-in developers are free to choose the `ContextPluginRuntime` subclass that best suits their particular context domain. The three subclasses of `ContextPluginRuntime` are presented in the following subsections.

5.1.1. The PushContextPluginRuntime Class

A `PushContextPluginRuntime` is a type of `ContextPluginRuntime` that performs *continuous context modeling* for a specific set of context information types, while broadcasting context events to all Dynamix applications holding an associated context subscription (and appropriate security credentials).

`PushContextPluginRuntime` implementations "push" context events to clients without requiring applications to specifically request a context scan. Instead, they operate independently in the background by modeling and broadcasting context information as deemed appropriate by the developer, and in accordance with the current power scheme (see section 5.4). As context information is modeled, context events are sent to the Dynamix Framework as described in section 5.3.1. Concrete Context Plug-in implementations override the `PushContextPluginRuntime`'s abstract methods, as described in Table 9.

PushContextPluginRuntime: Abstract Methods	
abstract void	doManualContextScan() Perform a single context scan when the runtime is STARTED and the <code>PluginPowerScheme</code> is set to MANUAL. Once the single scan is complete, if the <code>PluginPowerScheme</code> remains MANUAL, the <code>ContextPluginRuntime</code> should stop scanning context and wait until 'doManualContextScan' is called again, or another power scheme is chosen using 'setPowerScheme'.
abstract void	start() Starts continuous context acquisition and modeling, sending updated context events using the 'sendContextEvent' methods. Events generated by a <code>PushContextPluginRuntime</code> are always broadcast to all Dynamix applications holding an associated context subscription (and appropriate security credentials). Context modeling should continue until either 'stop' or 'destroy' are called. <ul style="list-style-type: none"> Note 1: This method may only be called when the <code>ContextPluginRuntime</code> is in <code>PluginState.INITIALIZED</code>. Note 2: Implementations should endeavor to conserve local resources and battery

	<p>power according to the current PluginPowerScheme (which may change over time).</p> <ul style="list-style-type: none"> • Note 3: If PluginPowerScheme.MANUAL is set, this method should not perform context scans until 'doManualContextScan' is called. • Note 4: This method will be called on a dedicated thread, so the method may block.
--	---

Table 9: Abstract PushContextPluginRuntime Methods

5.1.2. The PullContextPluginRuntime Class

The next type of ContextPluginRuntime subclass is the PullContextPluginRuntime, which performs *single* context modeling scans in response to the Dynamix Framework's requests to do so (using the 'handleContextRequest' method). PullContextPluginRuntime implementations "pull" context information from the environment in response to requests for context scans. Unlike PushContextPluginRuntimes, they do not operate independently in the background, but must operate in accordance with the current power scheme (see section 5.4). Requests for a context modeling scan are typically initiated by Dynamix-based applications and mediated by the Dynamix Framework. Responses to a context request are returned to the calling application (based on the original requestId) as described in section 5.3.2. Note that requesting applications must hold an appropriate context subscription and security credentials in order to request and receive context events from a PullContextPluginRuntime. Concrete Context Plug-in implementations override the PushContextPluginRuntime's abstract method, as described in Table 10.

PullContextPluginRuntime: Abstract Methods

<pre>abstract void</pre>	<p>handleContextRequest (UUID requestId, String contextDataType)</p> <p>Performs a single context scan for the specified requestId. Results for this method should be delivered using the standard 'sendContextEvent' methods, providing the originating requestId as the responseld (see section 5.3.2). This method may be called simultaneously by multiple threads, so care should be taken to always handle responses using the originating requestId of a given request thread. Importantly, if 'stop' or 'destroy' are called, the runtime MUST terminate all request processing immediately (and outstanding requests) according to the semantics defined in ContextPluginRuntime.</p> <ul style="list-style-type: none"> • Note 1: This method may only be called when the ContextPluginRuntime is in PluginState.STARTED. • Note 2: Implementations should endeavor to conserve local resources and battery power according to the current PluginPowerScheme (which may change over time). • Note 3: Implementations must queue multiple incoming context requests, handling them as first-come-first-serve • Note 4: PluginPowerScheme.MANUAL has no effect on IPullContextPluginRuntime, since each context scan is inherently manual. • Note 5: This method will be called on a dedicated thread, so the method may block.
--------------------------	--

Table 10: Abstract PullContextPluginRuntime Method

5.1.3. The PushPullContextPluginRuntime Class

The final type of ContextPluginRuntime subclass is the PushPullContextPluginRuntime, which combines both push and pull functionality. As with the PushContextPluginRuntime, PushPullContextPluginRuntime implementations "push" context events to clients without requiring applications to specifically request a context scan. In addition, like the PullContextPluginRuntime, PushPullContextPluginRuntime implementations also "pull" context information from the environment in response to requests for context scans. Hence, PushPullContextPluginRuntime must implement both push and pull context modeling operations in accordance with the current power scheme (see section 5.4). Context events are sent to applications as described in section 5.3.3. Concrete Context Plug-in implementations override the PushPullContextPluginRuntime's abstract methods, as described in Table 11.

PushPullContextPluginRuntime: Abstract Methods

abstract void	handleContextRequest (UUID requestId, String contextDataType) Performs a single context scan for the specified requestId. Results for this method should be delivered using the standard 'sendContextEvent' methods, providing the originating requestId as the responseId (see section 5.3.3). This method may be called simultaneously by multiple threads, so care should be taken to always handle responses using the originating requestId of a given request thread. Importantly, if 'stop' or 'destroy' are called, the runtime MUST terminate all request processing immediately (and outstanding requests) according to the semantics defined in ContextPluginRuntime. <ul style="list-style-type: none">• Note 1: This method may only be called when the ContextPluginRuntime is in PluginState.STARTED.• Note 2: Implementations should endeavor to conserve local resources and battery power according to the current PluginPowerScheme (which may change over time).• Note 3: Implementations must queue multiple incoming context requests, handling them as first-come-first-serve• Note 4: PluginPowerScheme.MANUAL has no effect on IPullContextPluginRuntime, since each context scan is inherently manual.• Note 5: This method will be called on a dedicated thread, so the method may block.
abstract void	doManualContextScan () Perform a single context scan when the runtime is STARTED and the PluginPowerScheme is set to MANUAL. Once the single scan is complete, if the PluginPowerScheme remains MANUAL, the ContextPluginRuntime should stop scanning context and wait until 'doManualContextScan' is called again, or another power scheme is chosen using 'setPowerScheme'.
abstract void	start () Starts continuous context acquisition and modeling, sending updated context events using the 'sendContextEvent' methods. Events generated by a PushContextPluginRuntime are always broadcast to all Dynamix applications holding an associated context subscription (and appropriate security credentials). Context modeling should continue until either 'stop' or

	<p>'destroy' are called.</p> <ul style="list-style-type: none"> • Note 1: This method may only be called when the ContextPluginRuntime is in PluginState.INITIALIZED. • Note 2: Implementations should endeavor to conserve local resources and battery power according to the current PluginPowerScheme (which may change over time). • Note 3: If PluginPowerScheme.MANUAL is set, this method should not perform context scans until 'doManualContextScan' is called. • Note 4: This method will be called on a dedicated thread, so the method may block.
--	---

Table 11: Abstract PushPullContextPluginRuntime Methods

5.2. Representing Context Information

Context Plug-ins represent contextual information modeled from the user's environment using implementations of the IContextInfo interface, which provides native, 'high resolution' contextual information appropriate to the given modeling domain. If needed, multiple IContextInfo types can be defined for a given Context Plug-in type. As IContextInfo objects are sent across process boundaries to applications, the interface extends Parcelable, which forces implementers to provide the necessary methods for serializing and de-serializing data according to the AIDL (Android Interface Definition Language). For details, please see: <http://developer.android.com/guide/developing/tools/aidl.html>. IContextInfo implementations can be arbitrarily complex (within the constraints of AIDL serialization). An overview of the methods provided by the IContextInfo interface is shown in Table 12.

IContextInfo: Method Summary

String	getContextType () Returns the type of the context information represented by the IContextInfo. This string must match one of the context information type strings described by the source ContextPlugin.
String	getStringRepresentation () Returns a string-based representation of the IContextInfo.
String	getStringRepresentationFormat () Returns a format type string that clients can use to select a proper parser for a given IContextInfo's string-based representation.
String	getImplementingClassname () Returns the fully qualified class-name of the class implementing the IContextInfo interface. Used when casting the IContextInfo entity to a concrete implementation.

Table 12: Overview of IContextInfo Methods

As shown above, the IContextInfo interface forces implementers to provide the fully qualified class-name of their implementing type through the **getImplementingClassname** method, which is used by capable clients to cast generic IContextInfo entities as concrete object types. The type of the context event can be

discovered through the **getContextType** method, which returns the type string of the IContextInfo entity. The IContextInfo interface also enables implementers to provide a string-based representation of their native contextual information. The string-based representation of the IContextInfo should be encoded using a well-known string format that is appropriate for the context domain. Clients use the **getStringRepresentationFormat** method to locate a suitable parser for string-based representations. The string data is used by clients who may not have access to the given IContextInfo classes on their class-path, but are still interested in working with string representations of the data. For example, a ContextEvent containing a java.util.Date may provide an ISO 8601 date format string that clients can parse without needing to have java.util.Date on their class-path. In this case, the string-based format type for the encoded ISO 8601 could be specified using a suitable Dublin Core Date element (e.g. 'http://purl.org/dc/elements/1.1/date').

5.3. Sending Context Events

Once context information has been extracted from the environment and encoded into an IContextInfo entity, it must be sent from a Context Plug-in runtime to the Dynamix Framework as a ContextEvent object. The key methods of the ContextEvent object are shown in Table 13.

ContextEvent: Key Methods	
String	getContextType() Returns the type of the context information represented by the event. This string must match one of the context information type strings described by the source ContextPlugin.
IContextInfo	getEventInfo() Returns the event's IContextInfo (if present); or null.
ContextPlugin Information	getEventSource() Returns information about the plug-in that generated the event.
String	getStringRepresentation() Returns a string-based representation of the IContextInfo's data type.
String	getStringRepresentationFormat() Returns the data type of the context representation available from getStringRepresentation() .

Table 13: ContextEvent Key Methods

Context events indicate their expiration information by extending the Expirable base class. An overview of the key methods of the Expirable class is shown in Table 14.

Expirable: Key Methods

boolean	expires() Returns true if the object expires; false otherwise.
Date	getExpireTime() Returns the expiration time, which is calculated by adding the specified expiration milliseconds to the event's time-stamp.
Date	getTimeStamp() Returns the time this expirable was generated.

Table 14: Expirable Key Methods

To construct a context event, the `IContextInfo` objects resulting from a context scan are first associated with a `FidelityLevel` using an object of type `SecuredContextInfo`. A `FidelityLevel` provides a categorization of the `IContextInfo`'s contextual precision (for use in Context Firewall provisioning). The notion of 'precision' is specific to a given context modeling domain and must be specified by the context domain expert developing a given Context Plug-in. In general, low fidelity implies the presence of `IContextInfo` with a relatively low level of detail as compared to max fidelity, which would provide context information with maximum precision or capability. For example, context information related to 'location' might be categorized as medium fidelity if determined by cell tower radio signals, and max fidelity if determined by GPS hardware. An overview of the available `FidelityLevel` types is shown in Table 15.

Field Summary

static <code>FidelityLevel</code>	LOW Flag specifying Context information of low fidelity.
static <code>FidelityLevel</code>	MEDIUM Flag specifying Context information of medium fidelity.
static <code>FidelityLevel</code>	HIGH Flag specifying Context information of high fidelity.
static <code>FidelityLevel</code>	MAX Flag specifying Context information of maximum fidelity.

Table 15: Overview of `FidelityLevel` Types

Once a `SecuredContextInfo` entity (or entities) has been created for a given context scan, it can then be sent to the Dynamix Framework using one of the utility methods included with each `ContextPluginRuntime` type. The following sections describe the available event sending utility methods.

5.3.1. PushContextPluginRuntime Event Utility Methods

As described in section 5.1.1, a PushContextPluginRuntime performs *continuous context modeling* for a specific set of context information types, while broadcasting context events to all Dynamix applications holding an associated context subscription (and appropriate security credentials). As IContextInfo entities are generated, the PushContextPluginRuntime can use one of several utility methods to send context events to the Dynamix Framework. The PushContextPluginRuntime's event utility methods are of two basic forms: expiring and non-expiring. In the expiring form, SecuredContextInfo entities can be sent to Dynamix with an indication of how long the context events are valid (in milliseconds). This expiration time will be included in the context events sent to clients and used by the Dynamix Framework for caching purposes. In the non-expiring form, SecuredContextInfo entities can be sent to Dynamix *without* an indication of how long the context events are valid. In this case, no expiration time will be included in the context events sent to clients and the Dynamix Framework will cache the event indefinitely (or until its caching policy causes the event to be removed). Table 16 provides an overview of these methods.

PushContextPluginRuntime: Event Utility Methods	
void	sendContextEvent (List<SecuredContextInfo> info) Called by subclasses in order to send non-expiring context events to the Dynamix Framework.
void	sendContextEvent (List<SecuredContextInfo> info, int expireMills) Called by subclasses in order to send expiring context events to the Dynamix Framework.
void	sendContextEvent (SecuredContextInfo data) Called by subclasses in order to send a non-expiring context event to the Dynamix Framework.
void	sendContextEvent (SecuredContextInfo data, int expireMills) Called by subclasses in order to send an expiring context event to the Dynamix Framework.

Table 16: PushContextPluginRuntime Event Utility Methods

5.3.2. PullContextPluginRuntime Event Utility Methods

As described in section 5.1.2, a PullContextPluginRuntime performs *single* context modeling scans in response to the Dynamix Framework's requests to do so (using the 'handleContextRequest' method). Responses to a context request are returned to the calling application (based on the requestId) through the Dynamix Framework using one of several utility methods shown in Table 17.

PullContextPluginRuntime: Event Utility Methods	
void	sendContextEvent (UUID responseId, List<SecuredContextInfo> info) Called by subclasses in order to send non-expiring context events to the Dynamix Framework.
void	sendContextEvent (UUID responseId, List<SecuredContextInfo> info, int expireMills)

	Called by subclasses in order to send expiring context events to the Dynamix Framework.
void	sendContextEvent (UUID responseId, SecuredContextInfo data) Called by subclasses in order to send a non-expiring context event to the Dynamix Framework.
void	sendContextEvent (UUID responseId, SecuredContextInfo data, int expireMills) Called by subclasses in order to send an expiring context event to the Dynamix Framework.

Table 17: PullContextPluginRuntime Event Utility Methods

Requests for a context modeling scan are typically initiated by Dynamix-based applications and mediated by the Dynamix Framework. The PullContextPluginRuntime's event methods are of two basic forms: expiring and non-expiring. In addition, each utility method has a responseId parameter, which must correspond to the requestId that originated the context scan. In the expiring form, SecuredContextInfo entities can be sent to Dynamix with an indication of how long the context event is valid (in milliseconds). This expiration time will be included in the context events sent to the requesting client and used by the Dynamix Framework for caching purposes. In the non-expiring form, SecuredContextInfo entities can be sent to Dynamix *without* an indication of how long the context event is valid. In this case, no expiration time will be included in the context events sent to the requesting client and the Dynamix Framework will cache the event indefinitely (or until its caching policy causes the event to be removed).

5.3.3. PushPullContextPluginRuntime Event Utility Methods

As described in section 5.1.3, a PushPullContextPluginRuntime combines both push and pull functionality; hence, it includes the event utility methods from both PullContextPluginRuntime and PushPullContextPluginRuntime. Importantly, the event utility method used to send context events must correspond to the matching context modeling logic. In particular, in push scenarios context events must be sent using the PushContextPluginRuntime methods presented in section 5.3.1. In pull scenarios, context events must be sent using the PullContextPluginRuntime methods presented in section 5.3.2.

5.4. Power Management

The Dynamix Framework manages the power state of all installed Context Plug-ins by notifying their associated runtimes about its current power scheme both during runtime initialization (see section 4.3) and during runtime using the **setPowerScheme**(PluginPowerScheme scheme) method (introduced in section 3.2). Within the Dynamix Framework, the PluginPowerScheme class is used to indicate the various power states, which specify how a particular runtime should attempt to optimize its power consumption in relation to its context acquisition performance and modeling precision. Concrete runtime implementations are free to optimize their context modeling performance as needed for a given context modeling domain. An overview of the supported power schemes is shown in Table 18.

PluginPowerScheme: Field Summary	
static PluginPowerScheme	MANUAL Context modeling should occur manually, as triggered by the user.
static PluginPowerScheme	POWER_SAVER Context modeling should attempt to conserve power as much as possible, potentially at the expense of reduced performance or precision.
static PluginPowerScheme	BALANCED Context modeling should balance performance and power consumption.
static PluginPowerScheme	HIGH_PERFORMANCE Context modeling should attempt to provide maximum performance, even at the expense of increased power consumption.

Table 18: Overview of Plugin Power Scheme States

6. Context Plug-in Compilation and Packaging

In order for a Context Plug-in to be integrated within a Dynamix Framework instance, it must be compiled as an Android 2.x application and packaged as a Dynamix-configured OSGi Bundle. As Context Plug-ins execute within the Android runtime, they must be compiled into the compact Dalvik Executable (.dex) format, which is optimized for resource-constrained systems. All referenced external libraries must also be compiled into the Dalvik Executable (.dex) format and included within the Context Plug-in's Bundle as well. Context Plug-ins are able to load classes and libraries contained within their Bundle JAR (and any embedded JARs); however, Android-based layout resources, graphics, and associated (via R.java) will not be available for layout inflation because of Android class-path issues that arise due to OSGi. An overview of a typical Eclipse Context Plug-in development structure is shown in Figure 3.

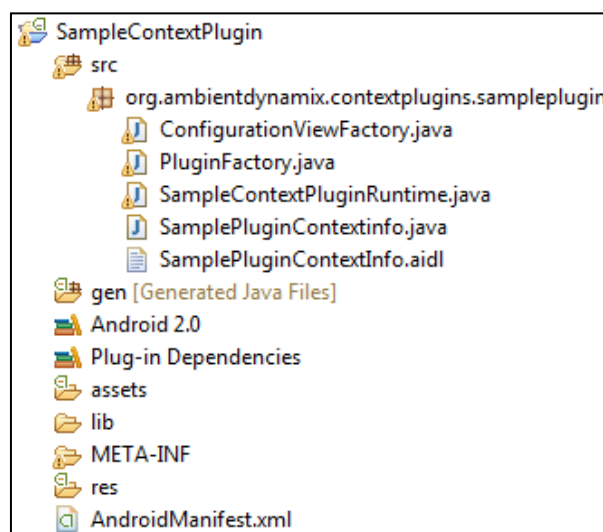


Figure 3: Sample Eclipse Context Plug-in Development Structure

Note the following regarding Figure 3:

- The MANIFEST.MF file exists within a META-INF folder. This directory structure must be maintained in the Bundle JAR.
- Each concrete IContextInfo class (e.g. SamplePluginContextInfo.java) must have a corresponding 'flag' file, which is named exactly the same, but has an aidl file extension (e.g. SamplePluginContextInfo.aidl). This file should contain the fully qualified class name of the IContextInfo class as a parcelable statement (see sample code).

During runtime, the following classes will be provided to Context Plug-ins by the Dynamix Framework:

- `org.ambientdynamix.application.api`
- `org.ambientdynamix.contextplugin.api`
- `org.ambientdynamix.contextplugin.api.security`

As such, these classes *should not be included within the Context Plug-in's Bundle JAR*. Aside from these packages, however, each Context Plug-in must be entirely self-contained (i.e. it cannot reference external libraries) and must be packaged within a *single* OSGi –compliant Bundle JAR, according to the Dynamix-specifications described in Table 19.

Context Plug-ins must be packaged as a Dynamix-configured OSGi Bundle¹¹. Briefly, an OSGi Bundle is a group of related Java classes (and other resources, such as software libraries or images), which are packaged within a standard Java JAR with additional OSGi metadata (included within the Bundle's MANIFEST.MF). Dynamix Context Plug-ins must be specified using the Dynamix-specific OSGi Bundle Manifest Specification shown in Table 19.

OSGi Entity	Specification
Manifest-Version:	1.0
Bundle-ManifestVersion:	2
Bundle-Name:	The text-based name of the plug-in.
Bundle-Vendor:	The text-based name of the plug-in vendor
Bundle-SymbolicName:	A globally unique plug-in identifier, which must match the plug-in id specified in its associated Context Plug-in Description Document (see section 7.4). This identifier is defined using a hierarchical naming pattern, with levels in the hierarchy separated by periods. In general, a plug-in identifier begins with the top level domain name of the organization and then the organization's domain and then any sub-domains listed in reverse order. Plug-in identifier should be lowercase.
Bundle-Version:	The version of the plug-in, using a <major>.<minor>.<micro> scheme.
Export-Package:	The Context Plug-in's exported packages.
Import-Package:	The Context Plug-in must import <code>org.ambientdynamix.application.api</code> , <code>org.ambientdynamix.contextplugin.api</code> and <code>org.ambientdynamix.contextplugin.api.security</code> .

Table 19: Dynamix-specific OSGi Bundle Manifest Specification

¹¹ <http://en.wikipedia.org/wiki/OSGi#Bundles>

Based on the Bundle Manifest Specification presented in Table 17, and example MANIFEST.MF file is shown below in Figure 4.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Sample Context Plugin
Bundle-Vendor: The Ambient Dynamix Project
Bundle-SymbolicName: org.ambientdynamix.contextplugins.sampleplugin
Bundle-Version: 1.0.0
Export-Package: org.ambientdynamix.contextplugins.sampleplugin
Import-Package: org.ambientdynamix.application.api,
    org.ambientdynamix.contextplugin.api,
    org.ambientdynamix.contextplugin.api.security
DynamicImport-Package: *
Bundle-ClassPath: .
```

Figure 4: Sample MANIFEST.MF File with a Context Plug-in OSGi Configuration

7. Context Plug-in Deployment

Once a Context Plug-in has been compiled and packaged, as described in section 6, it can be configured for deployment from a Dynamix network infrastructure (Dynamix Infrastructure), which consists of conventional Web servers that host a specific set of configuration files and plug-in JARs. The Dynamix Infrastructure is designed to be simple and highly scalable; hence, interactions are client driven, meaning that no dedicated programming logic is required on the server side. Since the Dynamix Infrastructure is based on standard Web technologies, common security techniques (e.g. HTTPS and Basic access authentication) and well-known scaling strategies (e.g. clustering, memcached¹², load balancing, etc.) can be used to support metadata and plug-in provisioning. The following sections describe the Dynamix Infrastructure with a focus on Context Plug-in deployment.

7.1. Overview of the Dynamix Infrastructure

A Dynamix Infrastructure is comprised of distributed Dynamix clients and three server types: bootstrap servers, plug-in repository servers and content servers. Bootstrap servers host Dynamix Core Configuration (ACC) Documents, which provide Dynamix clients global state information and a list of available plug-in repository servers (see 7.3). Plug-in repository servers host a set of Plug-in Description Documents, which describe the various plug-ins that are available for runtime integration (see 0). Finally, content servers host the plug-in Bundle JARs, Dynamix Framework update binaries and other comparably ‘large’ content. Based on these three server types, a Dynamix Infrastructure can be arranged to support a variety of scenarios. For example, in simple scenarios (e.g. development), a single physical server may be used as all three server types. In more complex scenarios, the bootstrap servers and plug-in repository servers may be clustered on several machines (supported by load balancing) while the plug-in Bundle JARs are hosted on a content delivery network (CDN).

¹² <http://memcached.org/>

7.2. Dynamix Infrastructure Interaction

Communication within a Dynamix Infrastructure is based on a simple interaction protocol that is made up of server side configuration files and client-side logic. Dynamix clients download appropriate configuration files, parse the contents and utilize the embedded metadata to determine global state information and discover plug-ins that are available for runtime integration. Dynamix clients make *local* decisions about the suitability of discovered plug-ins based on local capability analysis, user interaction and the context modeling functionality requested by Dynamix-based applications. Dynamix client users control which plug-ins are installed into a Dynamix device, the permissions granted to each, and the type and fidelity of the context information provisioned to Dynamix applications. Communication between a Dynamix Client and a Dynamix Infrastructure follows the interaction scheme outlined below.

1. A Dynamix client is configured with a specific Dynamix bootstrap server (and an optional backup bootstrap server to support automatic failover). For details on Dynamix client configuration, see section 8.1.
2. During runtime, a Dynamix client contacts its bootstrap server and downloads the current Dynamix Core Configuration (ACC), which provides global state information and a list of available plug-in repository servers. The ACC includes an expiration time, which indicates to the client when a new version of the ACC should be downloaded.
3. The Dynamix client parses the ACC to discover the available plug-in repository servers that are compatible with the client's platform (e.g. Android) and Dynamix version. Plug-in Repositories host information about available plug-ins that can be integrated on demand into a Dynamix client.
4. The Dynamix client selects suitable plug-in repository servers based on the platform, version and plug-in type information embedded in the ACC.
5. The Dynamix client contacts each compatible plug-in repository server and downloads its associated list of Plug-in Description Documents (PDD) for a specific type of plug-in the client is interested in. Broadly, a PDD provides detailed metadata regarding available plug-ins, including plug-in identifiers, capability dependencies, required permissions, installation URLs, etc.
6. The Dynamix client builds a list of candidate plug-ins for installation by filtering out incompatible plug-ins from the downloaded PDDs. Filtering is performed based on the versioning information, identifiers and dependencies listed in the PDD (e.g. as the result of local capability analysis).
7. Based on specific user requests and/or application context subscription requests, the metadata from the filtered PDD is used to guide plug-in selection, download and integration during runtime.
8. During installation, a Dynamix client downloads selected plug-in Bundles from the content server addresses listed in the filtered PDD.
9. The Dynamix client continually refreshes its list of available plug-ins using the expiration information listed in the ACC. The continually updated list of plug-ins is used to integrate suitable plug-ins as needed by users and applications.

7.3. The Dynamix Bootstrap Process

As described above, a Dynamix client begins an interaction with the Dynamix Infrastructure by contacting its configured bootstrap server. A Dynamix bootstrap server is a conventional Web server that hosts one or more Dynamix Bootstrap Documents (DBDs). Broadly, an DBD specifies global configuration options and provides a listing of the available Dynamix plug-in repository servers, as shown in Figure 7. A detailed overview of the Dynamix Core Configuration Schema is provided in section 9.

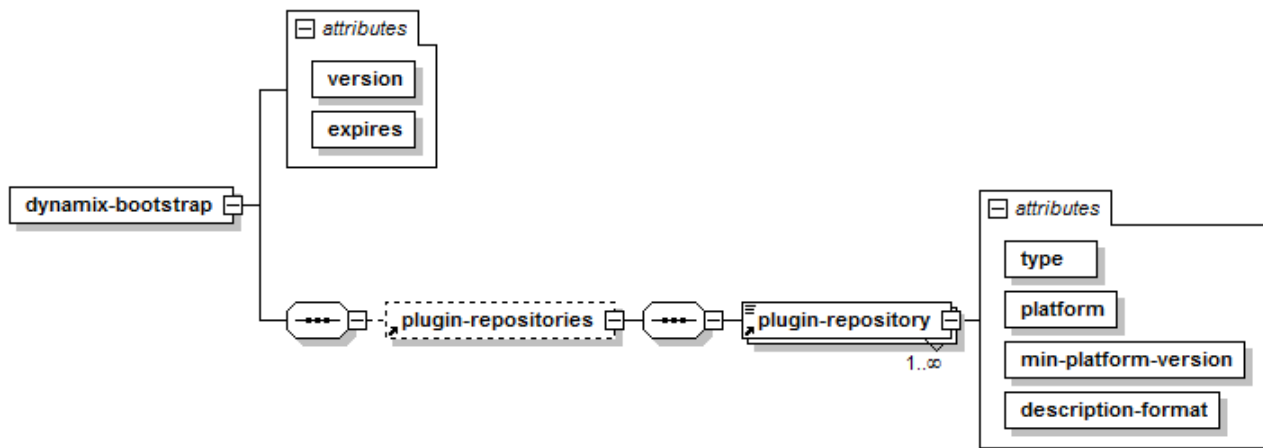


Figure 5: Dynamix Bootstrap Schema Version 1.0.0

Based on the schema shown in Figure 5, an example ACC XML file is shown in Figure 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<dynamix-bootstrap version="1.0.0" expires="2011-07-16T19:20+01:00">
  <plugin-repositories>
    <plugin-repository
      type="context-plugin"
      description-format="1.0.0"
      platform="android"
      min-platform-version="2.0">
      https://user:pass@ambientdynamix.org/android/2.1/c-plugs.xml
    </plugin-repository>
  </plugin-repositories>
</dynamix-bootstrap>
```

Figure 6: Example Dynamix Bootstrap Document

With reference to the example Dynamix Bootstrap Document shown above, note the following:

- The DBD version number is 1.0. This number is used by the Dynamix client to find a suitable parser.
- The DBD expires on 2011-07-16 at 19:20+01:00. After this time, the Dynamix client must contact the bootstrap server for a new DBD.
- In this case, there is a single supported platform type, which is 'android'; however, there may be many platforms depending on the Dynamix Infrastructure deployment.
- In this case, there is a single plug-in repository, which hosts plug-ins of type "context-plugin", describes plug-ins using a version 1.0.0 description format (described shortly), requires Dynamix Framework version of 0.6.0 or greater, and requires Android platform version of 2.0 or greater.
- The Plug-in repository URL includes a HTTPS scheme, username and password (optional), and a logically arranged network path based on platform versioning.

7.4. Plug-in Repository Interaction

After a Dynamix client downloads and parses the current Dynamix Bootstrap Document (DBD) from its bootstrap server, it builds a list of compatible Plug-in Repositories based on the plug-in type, target platform and Dynamix version information contained within the DBD. At the completion of this list-building process, the Dynamix client contacts each compatible Plug-in Repository and downloads their associated Plug-in Description Documents (PDD). A PDD can be of various types, depending on the plug-in type described. For Context Plug-in development, the associated PDD type is the Context Plug-in Description (CPD) format. Briefly, the CPD is an XML structure that provides Dynamix clients information about the Context Plug-ins that are available from a specific Plug-in Repository for a given target platform and Dynamix version. The information contained in a CPD includes plug-in identifiers, supported context information types, capability dependencies, required permissions, installation URLs, etc. The Context Plug-in Description Schema is shown in Figure 7. A detailed overview of the Context Plug-in Description Schema is provided in section 10.

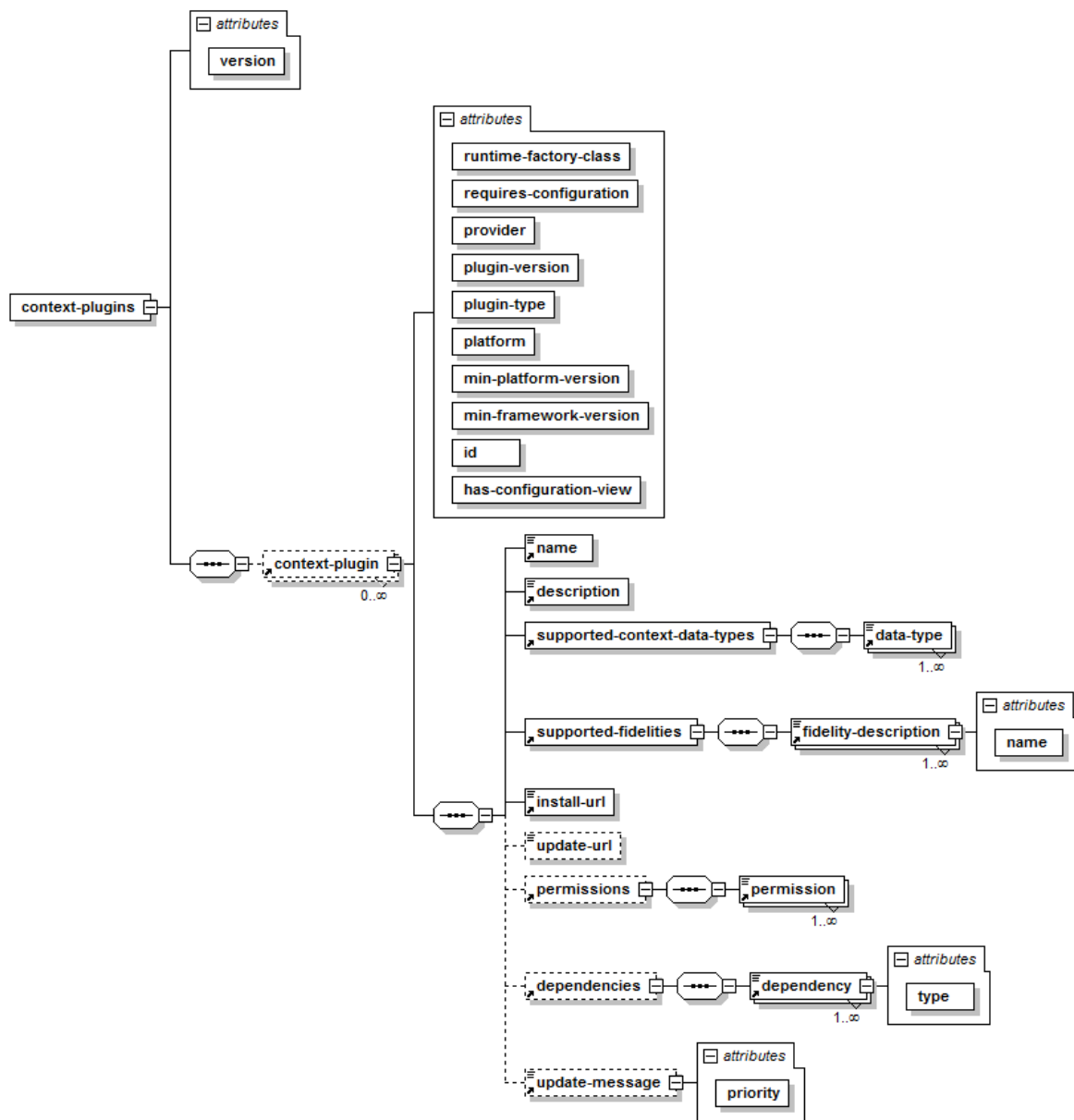


Figure 7: The Context Plug-in Description Schema version 1.0.0

Based on the schema shown in Figure 7, an example CPD document is shown in Figure 8.

```
<?xml version="1.0" encoding="UTF-8"?>
<context-plugins version="1.0.0">
  <context-plugin id="org.test.sampleplugin"
    plugin-version="1.0.0"
    provider="The Ambient Dynamix Project"
    plugin-type="PUSH"
    platform="android"
    min-platform-version="2.0"
    min-framework-version="0.7.0"
    requires-configuration="true"
    has-configuration-view="true"
    runtime-factory-class="org.test.PluginFactory">
    <name>Sample Context Plugin</name>
    <description>Sample Context Plugin description text.</description>
    <supported-fidelities>
      <fidelity-description name="MAX">Description for MAX fidelity.
    </fidelity-description>
      <fidelity-description name="HIGH">Description for HIGH fidelity.
    </fidelity-description>
      <fidelity-description name="MEDIUM">Description for MEDIUM fidelity.
    </fidelity-description>
      <fidelity-description name="LOW">Description for LOW fidelity.
    </fidelity-description>
    </supported-fidelities>
    <supported-context-data-types>
      <data-type>org.test.sample_type</data-type>
    </supported-context-data-types>
    <permissions>
      <permission>Context.INPUT_METHOD_SERVICE</permission>
      <permission>Context.WINDOW_SERVICE</permission>
    </permissions>
    <dependencies>
      <dependency type="android.feature">
        LocationManager.GPS_PROVIDER</dependency>
      </dependencies>
    <install-url>https://user:pass@test.org/org.test.sampleplugin_1.0.0.jar
    </install-url>
    <update-message priority="OPTIONAL">Update message</update-message>
  </context-plugin>
</context-plugins>
```

Figure 8: Example Context Plug-in Description Document

With reference to the example Context Plug-in Description document shown above, note the following:

- The document provides information about a single Context Plug-in named `Sample Context Plugin`. Note that the document could describe many Context Plug-ins in this way, each described similarly within the `'context-plugins'` XML element.
- The document provides a variety of metadata, such as id, name, version, target platforms, etc.
- The document provides the fully qualified class-name of its factory class (see section 4.2.1):
`org.test.PluginFactory`.
- The CPD includes additional details, such as the plug-in's supported context information types, fidelity levels, required permissions, install URL, etc.

8. Framework Integration and Testing

Once a Context Plug-in has been configured for deployment, as described in section 7, it can be dynamically integrated into one or more Dynamix clients for testing. The dynamic integration of plug-ins within a Dynamix client follows the Dynamix Infrastructure interaction scheme described in section 7.2. As described in section 7.3, a Dynamix client initiates a communication with its Dynamix Infrastructure by first contacting its configured bootstrap server. During the bootstrap process, the client downloads its Dynamix Bootstrap Document and discovers the available plug-in repository servers that are compatible with its platform and framework version. Next, the Dynamix client contacts a suitable plug-in repository server (or servers) and downloads its Plug-in Description Document (PDD) for a give plug-in type. Recall that for Context Plug-in deployment, the associated PDD type is the Context Plug-in Description (CPD) format, which was described in section 0. Once the Dynamix client builds a list of candidate plug-ins for installation (by filtering out incompatible plug-ins from the downloaded PDDs), it then uses the metadata from the filtered PDD is used to guide plug-in selection, download and integration during runtime. The following sections describe the Dynamix client configuration and setup.

8.1. Dynamix Client Configuration

A Dynamix client is bound with a particular Dynamix Infrastructure through a single configuration file on a Dynamix device, which specifies (among other things) the framework's primary and backup bootstrap servers. This configuration file must be named "dynamix.conf", must exist in the 'conf' subdirectory of the Dynamix Framework's 'app_data' directory, and must be readable by the Dynamix runtime process on the device. For details on configuring a Dynamix client, please see the example 'dynamix.conf' file accompanying this documentation. The relevant configuration snippet for bootstrap server configuration is shown below in Figure 9.

```
...
# Specifies if Dynamix is allowed to discover and dynamically install
# framework updates (e.g. plug-ins) at runtime
allow.framework.updates=true

# Dynamix bootstrap server details
# Optional if allow.framework.updates=false
# Backup server is optional
# ~~~~~
primary.bootstrap.server.alias= Example Dynamix Primary Bootstrap Server
primary.bootstrap.server.url=http://ambientdynamix.org/bootstrap.xml
primary.bootstrap.server.port=80
primary.bootstrap.server.username=
primary.bootstrap.server.password=
...
```

Figure 9: Dynamix Configuration Snippet

Note the following regarding Figure 9:

- `allow.framework.updates` must be set to true.
- The primary Dynamix bootstrap server must be specified; however, the backup is optional.

8.2. Dynamix Client Setup

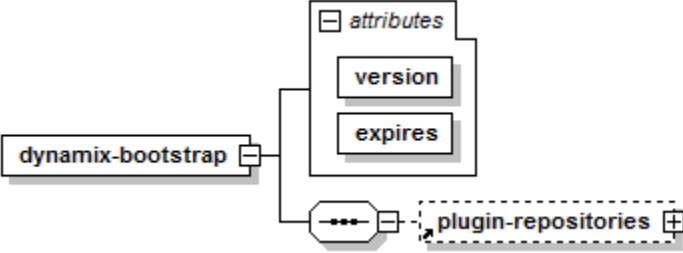
TODO

8.3. Plug-in Integration and Testing

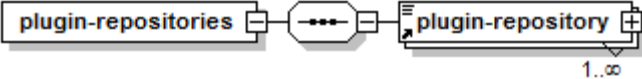
TODO

9. Appendix A: The Dynamix Bootstrap Schema

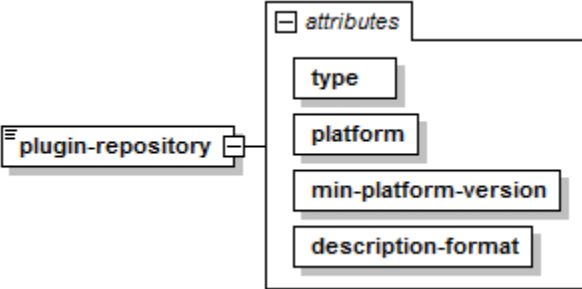
element **dynamix-bootstrap**

diagram						
properties	content complex					
children	<u>plugin-repositories</u>					
attributes	Name <u>version</u> <u>expires</u>	Type derived by: xs:string derived by: xs:string	Use required required	Default	Fixed	annotation

element **plugin-repositories**

diagram						
properties	content complex					
children	<u>plugin-repository</u>					
used by	element <u>dynamix-bootstrap</u>					

element **plugin-repository**

diagram						
type	extension of <u>ST plugin-repository</u>					
properties	content complex					
used by	element <u>plugin-repositories</u>					
facets	enumeration http://dcarlson.info/dynamix/updates-dcarlson.xml					
attributes	Name <u>type</u> <u>platform</u> <u>min-platform-version</u> <u>description-format</u>	Type derived by: xs:string derived by: xs:string derived by: xs:decimal derived by: xs:string	Use required required required required	Default	Fixed	annotation

10. Appendix B: The Context Plug-in Description Schema

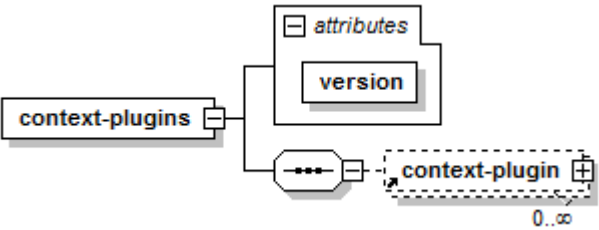
element **context-plugin**

diagram						
properties	content complex					
children	<u>name</u> <u>description</u> <u>supported-context-data-types</u> <u>supported-fidelities</u> <u>install-url</u> <u>update-url</u> <u>permissions</u> <u>dependencies</u> <u>update-message</u>					
used by	element <u>context-plugins</u>					
attributes	Name runtime-factory-class requires-configuration provider plugin-version plugin-type platform min-platform-version min-framework-version id has-configuration-view	Type xs:string xs:boolean derived by: xs:string derived by: xs:string derived by: xs:string derived by: xs:string derived by: xs:decimal derived by: xs:string derived by: xs:string xs:boolean	Use required required required required required required required required	Default 	Fixed 	annotation


element **context-plugin/update-url**

diagram	
type	xs:anyURI
properties	isRef 0 minOcc 0 maxOcc 1 content simple

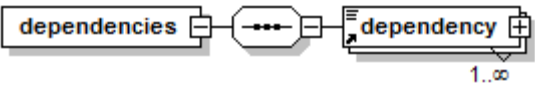
element **context-plugins**

diagram						
properties	content complex					
children	<u>context-plugin</u>					
attributes	Name <u>version</u>	Type derived by: xs:string	Use required	Default	Fixed	annotation

element **data-type**

diagram	
type	restriction of xs:string
properties	content simple
used by	element <u>supported-context-data-types</u>

element **dependencies**


diagram						
properties	content complex					
children	<u>dependency</u>					
used by	element <u>context-plugin</u>					

element **dependency**


diagram						
---------	---	--	--	--	--	--

type	extension of ST_dependency					
properties	content complex					
used by	element dependencies					
facets	enumeration LocationManager.GPS_PROVIDER					
attributes	Name <u>type</u>	Type derived by: xs:string	Use required	Default	Fixed	annotation

element **description**

diagram						
type	xs:string					
properties	content simple					
used by	element context-plugin					


element **fidelity-description**

diagram						
type	extension of ST_fidelity-description					
properties	content complex					
used by	element supported-fidelities					
facets	enumeration Description for HIGH fidelity. enumeration Description for LOW fidelity. enumeration Description for MAX fidelity. enumeration Description for MEDIUM fidelity. enumeration Description for NONE fidelity.					
attributes	Name <u>name</u>	Type derived by: xs:string	Use required	Default	Fixed	annotation

element **install-url**


diagram						
type	xs:anyURI					
properties	content simple					
used by	element context-plugin					

element **name**

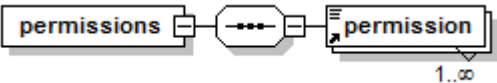
diagram						
type	restriction of xs:string					
properties	content simple					

used by	element context-plugin
---------	--

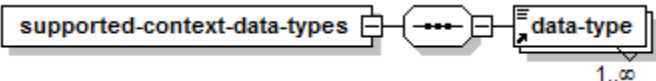
element **permission**

diagram	
type	restriction of xs:string
properties	content simple
used by	element permissions

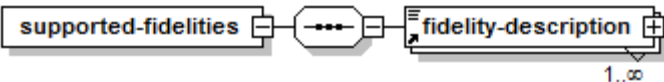
element **permissions**

diagram	
properties	content complex
children	permission
used by	element context-plugin

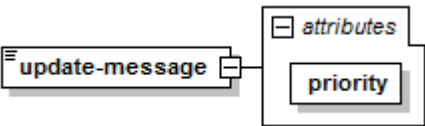
element **supported-context-data-types**

diagram	
properties	content complex
children	data-type
used by	element context-plugin

element **supported-fidelities**

diagram	
properties	content complex
children	fidelity-description
used by	element context-plugin

element **update-message**

diagram	
type	extension of ST update-message
properties	content complex
used by	element context-plugin

facets	enumeration Update message text.					
attributes	Name <u>priority</u>	Type derived by: xs:string	Use required	Default	Fixed	annotation