APPLICATION DEVELOPMENT GUIDE

For the Dynamix Framework v0.9.0 ***Dr.-Ing. Darren Carlson***
Document: v1.0.0

1. Introduction

The Dynamix Framework is an open-source context modeling middleware designed to help simplify context-aware software development on resource-constrained mobile devices. 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 dynamic 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 Dynamix application development process, including the Dynamix Framework's Facade application programming interface (API) and associated event model. It also covers related topics, such as context information representation and application development tips. This document does not describe the Dynamix Framework architecture in detail or the Context Plug-in development process. For further details, please see the documentation available at TODO.

1.2. License

1.3. Table of Contents

2. Dynamix Framework 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 control mechanisms for acquiring and controlling it. As a result, developers often spend considerable time and expense inventing (or re-inventing) the foundational context modeling mechanisms needed to support their applications.

Dynamix addresses the increasing complexity of context acquisition, modeling 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 modeling capabilities, and more.

The actual work of context modeling 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.

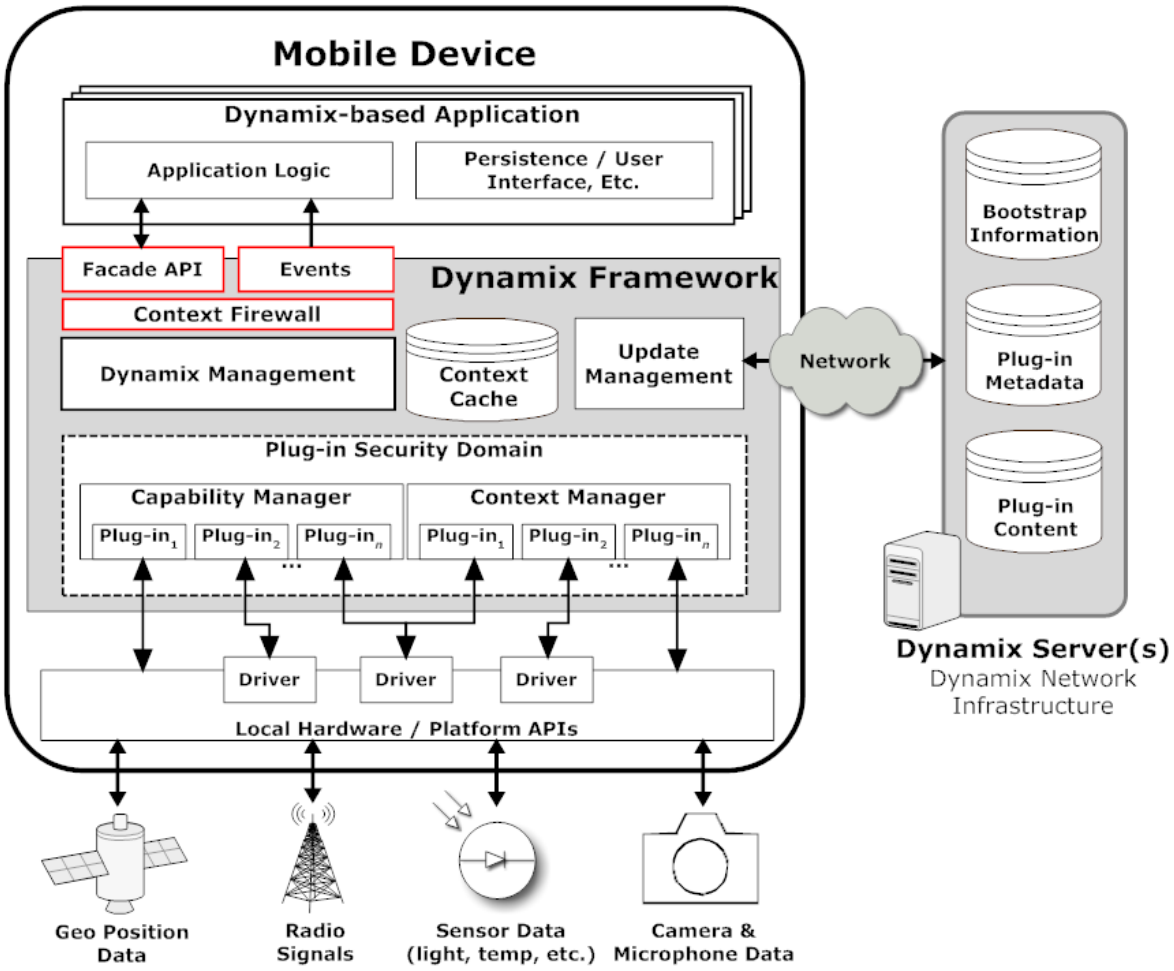

Figure 1: Overview of the Dynamix Framework

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

The Android platform was chosen as the initial deployment target for the Dynamix Framework. As such, Dynamix is designed as an Android Service[1], 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.

2.3. Dynamix Application Overview

Dynamix application development consists of a set of related tasks, which are designed to structure an application so that it can issue commands and receive events from a Dynamix Service. An overview of these tasks and their related document sections are shown in Table 1.

Development Task	Description
1. Create an Android Application	Android application development is not covered in this guide. See http://developer.android.com/guide/topics/fundamentals.html
2. Connect the Application to the Dynamix Service See section: 2.4	Use Android's AIDL mechanisms to connect the Android application to the local Dynamix Framework instance (running as an Android Service in a separate process).
3. Implement the Dynamix Façade and Event Model Interfaces. See section: 3	Using utility classes from the DynamixApplicationApi, implement the IDynamixFacade and the IDynamixListener interfaces. This provides the foundation for interacting with the local Dynamix Framework.
4. Handle Context Scanning and Context Events See section: 4	Use the Dynamix Façade to control the context modeling processes for the given application. Respond to incoming context events as needed.

Table 1: Overview of Dynamix Application Development Tasks

Dynamix applications are standard Android applications that incorporate extra context modeling functionality provided by a local Dynamix Framework instance (Dynamix Service). As such, Dynamix-based applications are developed using the conventional Android application model[2]. To receive contextual information from Dynamix, an application uses the Dynamix Service's Façade and Event Model, which is comprised of two related AIDL (Android Interface Definition Language)[3] interfaces: the IDynamixFacade and the IDynamixListener (available in the Dynamix Application Api, which can be downloaded from TODO). Briefly, the IDynamixFacade enables applications to register to receive context events, setup context subscriptions, request context scans, etc. The IDynamixListener interface is used to listen for status updates and context events from the Dynamix Service. Status updates include information about the framework itself

[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

(e.g. active or inactive) and about the results of commands issues via the IDynamixFacade interface (e.g. context subscription added). Context events include contextual information provided by the Dynamix Service's context acquisition and modeling processes. Each context event contains native context information (as an instance of IContextInfo, see section 4.3), the time the event was generated and how long the event is valid (including forever). Each event also provides a string-based representation of the context information, to support a broad range of clients, including those incapable of parsing the native IContextInfo instance. Dynamix applications are free to use context events to adapt or optimize their functionality as needed. As the Dynamix approach is domain neutral, the type functionality provided in response to context events is entirely up to the application.

2.4. Connecting to the Dynamix Service

As previously introduced, a Dynamix Framework instance (Dynamix Service) runs in the background on a user's device as an Android Service; modeling contextual information from the environment using a dynamically installed set of Context Plug-ins. Dynamix applications interact with a Dynamix Service using its Façade and event model, which are provided by the IDynamixFacade and the IDynamixListener interfaces respectively. Before the Dynamix Façade can be used, however, a Dynamix application must first connect to the Dynamix Service using an Android ServiceConnection[4]. To support the creation of a ServiceConnection, the Dynamix Application Api (see TODO) includes an IDynamixFacade.sub and IDynamixListener.stub, which provide the "plumbing" necessary to establish a communication channel between the application and the Dynamix Service. A ServiceConnection between a Dynamix application and a Dynamix Service is created as follows[5]. (For details, please see the application development quick-start guide and example source-code.)

1. Include the latest Dynamix Application Api JAR file in the project's build path.
2. Declare an instance of the `IDynamixFacade` interface.
3. Create an instance of type `IDynamixListener` using "`new IDynamixListener.Stub();`"
4. Implement each of the `IDynamixListener` methods as needed by the application.
5. Implement the `ServiceConnection` methods to handle the communication channel.
6. Call `Context.bindService()`, passing in your `ServiceConnection` implementation.
7. In your implementation of `onServiceConnected()`, you will receive an `IBinder` instance (called `service`). Call *`IDynamixFacade`*`.Stub.asInterface((IBinder)`*`service`*`)` to cast the returned parameter to the *`IDynamixFacade`* type.
8. Register the previously created `IDynamixListener` class as a Dynamix listener using the `IDynamixFacade's addDynamixListener` method.
9. Call the methods defined on the IDynamixFacade interface. You should always trap `DeadObjectException` exceptions, which are thrown when the connection has broken; this will be the only exception thrown by remote methods.
10. To disconnect, call `Context.unbindService()` with the instance of your interface.

[4] http://developer.android.com/reference/android/content/ServiceConnection.html
[5] Adapted from http://developer.android.com/guide/developing/tools/aidl.html#Calling

3. Foundations of the Dynamix Façade and Event Model

After a Dynamix application establishes a ServiceConnection with the Dynamix Service, as described in section 2.4, the IDynamixFacade and the IDynamixListener interfaces can be used to interact with the Dynamix Framework. These interfaces are presented next.

3.1. The IDynamixFacade Interface

The IDynamixFacade is the interface that applications use to communicate with the Dynamix Service. As described in section 2.4, a Dynamix application obtains a reference to an IDynamixFacade during the creation of a ServiceConnection with the Dynamix Service. The IDynamixFacade is shown in Table 2. For a detailed overview of these methods, see: http://dynamixframework.org/javadocs/

Method Summary

void	**addContextSubscription**(IDynamixListener listener, String contextType) Adds a context subscription for the specified listener and context type, indicating that the listener is interested in receiving events when Dynamix detects that particular type of context info.
void	**addDynamixListener**(IDynamixListener listener) Adds the IDynamixListener to the Dynamix Framework.
void	**closeSession**() Immediately closes the application's Dynamis session, removing all of the application's context subscriptions.
List<ContextPluginInformation>	**getContextPluginInformation**() Returns a List of type ContextPluginInformation, which describes the currently installed ContextPlugins within the Dynamix Framework along with the supported context types for each.
List<String>	**getContextSubscriptions**(IDynamixListener listener) Returns the context subscriptions that have been registered by the specified IDynamixListener.
DeviceInfo	**getDeviceInfo**() Returns a DeviceInfo that represents the current state of the device running the Dynamix Framework.
String	**getListenerId**(IDynamixListener listener) Returns the listener's id.

String	**getSessionId**() Returns the session id for this application, which is used for some secure interactions with Dynamix, such as launching context acquisition interfaces for Context Plug-ins of type pull interactive.
boolean	**isSessionOpen**() Returns true if the application's session is open; false otherwise.
void	**openSession**() Indicates that the calling application wishes to open a session with the Dynamix framework.
void	**removeAllContextSubscriptions**() Removes all previously added context subscriptions for the application for all listeners, regardless of contextType.
void	**removeContextSubscription**(IDynamixListener listener, String contextType) Removes a previously added context subscription for the specified listener and contextType.
void	**removeContextSubscriptions**(IDynamixListener listener) Removes all previously added context subscription for the specified listener, regardless of contextType.
void	**removeDynamixListener**(IDynamixListener listener) Removes the IDynamixListener from the Dynamix Framework.
String	**requestConfiguredContextScan**(IDynamixListener listener, String pluginId, String contextType, android.os.Bundle scanConfig) Requests that Dynamix perform a dedicated context scan using the specified plugin, contextType and scanConfig.
boolean	**requestContextPluginInstallation**(String url) Request that Dynamix install a specific ContextPlugin on behalf of the Application.
String	**requestContextScan**(IDynamixListener listener, String pluginId, String contextType) Requests that Dynamix perform a dedicated context scan using the specified plugin and contextType.
void	**resendAllCachedContextEvents**(IDynamixListener listener)

	Resends all ContextEvents that have been cached by Dynamix for the specified listener.
void	**resendAllTypedCachedContextEvents**(IDynamixListener listener, String contextType) Resends all ContextEvents (of the specified contextType) that have been cached by Dynamix for the specified listener.
void	**resendCachedContextEvents**(IDynamixListener listener, int previousMills) Resends the ContextEvent entities that have been cached for the listener within the specified past number of milliseconds.
void	**resendTypedCachedContextEvents**(IDynamixListener listener, String contextType, int previousMills) Resends the ContextEvent entities (of the specified contextType) that have been cached for the listener within the specified past number of milliseconds.

Table 2: The IDynamixFacade Interface

3.2. IDynamixListener Events

Responses to asynchronous IDynamixFacade methods, along with other Dynamix Service status updates, are provided to applications through the events defined in the IDynamixListener interface. Dynamix applications create an instance of the IDynamixListener interface using the IDynamixListener.Stub class, as described in section 2.4. An overview of the various IDynamixListener events is shown in Table 3. For a detailed overview of these events, see: http://dynamixframework.org/javadocs/

Method Summary	
void	**onAwaitingSecurityAuthorization**() Notification that the application is awaiting security authorization by the Dynamix Framework.
void	**onContextEvent**(ContextEvent event) Notification of an incoming ContextEvent.
void	**onContextPluginInstalled**(ContextPluginInformation plugin) Notifies the application that a new Context Plug-in has been installed.
void	**onContextPluginInstallFailed**(ContextPluginInformation plug, String message) Notifies the application that a Context Plug-in has failed to install.
void	**onContextPluginUninstalled**(ContextPluginInformation plugin) Notifies the application that a previously installed Context Plug-in has been uninstalled.

void	**onContextScanFailed**(String requestId, String message) Notifies the application that a previously requested context scan has failed.
void	**onContextSubscriptionAdded**(ContextPluginInformation plugin, String contextType) Notifies the listener that a context subscription for the given context type has been added.
void	**onContextSubscriptionRemoved**(String contextType) Notifies the listener that a context subscription for the given context type has been removed.
void	**onContextTypeNotSupported**(String contextType) Notifies the application that the requested context type is not supported, and that the requested context subscription was not added.
void	**onDynamixListenerAdded**(String listenerId) A response to the IDynamixFacade 'addDynamixListener' method indicating that the Dynamix listener has been added.
void	**onDynamixListenerRemoved**() A response to the IDynamixFacade 'removeDynamixListener' method indicating that the Dynamix listener has been removed.
void	**onInstallingContextPlugin**(ContextPluginInformation plugin) Notifies the application that a new Context Plug-in is being installed.
void	**onInstallingContextSupport**(ContextPluginInformation plugin, String contextType) Notifies the application that a plugin installation has begun for the specified contextType.
void	**onSecurityAuthorizationGranted**() Notification that the application has been granted security authorization by the Dynamix Framework.
void	**onSecurityAuthorizationRevoked**() Notification that the application's security authorization has been revoked by the Dynamix Framework.
void	**onSessionClosed**() Notification that the Dynamix session has been closed.
void	**onSessionOpened**(String sessionId) Notification that the Dynamix session has been opened.

Table 3: IDynamixListener Events

3.3. Basic Dynamix Interaction

After a Dynamix application establishes a ServiceConnection with the Dynamix Service, as described in section 2.4, an application can interact with Dynamix using the methods of the IDynamixFacade (see section 3.1), and the events specified in IDynamixListener (see section 3.2). First, an application registers one (or more) Dynamix listeners using the **`addDynamixListener`** method. After Dynamix has added the specified listeners, each listener will receive an **`onDynamixListenerAdded`** event. Next, the application opens a session with the Dynamix Service using the **`openSession`** method. During session initiation, the Dynamix Service validates the application's identity and checks for an associated Context Firewall policy. Importantly, Dynamix applications *must* have a Context Firewall policy defined by the user in order to receive security authorization from the Dynamix Service. A Context Firewall policy allows the user to precisely manage the type and fidelity of the contextual information provisioned to each Dynamix application. If no policy exists, Dynamix notifies the user, who must create a policy before the application is able to interact further with the Dynamix Service. Applications that do not have security authorization receive an **`onAwaitingSecurityAuthorization`** event. Once security authorization has been granted for the application, the **`onSecurityAuthorizationGranted`** event is raised, meaning that the application is allowed to call additional methods on the Dynamix Service (if the service is active). After receiving the **`onSecurityAuthorizationGranted`** event, the application will receive either an **`onSessionOpened`** event or an **`onSessionClosed`** event, depending on the active state of the Dynamix Service. (Note that a user can revoke an application's security authorization at any time. If this happens, the application will receive the **`onSecurityAuthorizationRevoked`** event.)

An *open* Dynamix session is actively modeling context information from the user's environment using its set of dynamically installed Context Plug-ins. A *closed* Dynamix session is not modeling contextual information, but still maintains application connection state. If the **`onSessionOpened`** is raised, the application is free to call additional IDynamixFacade methods (e.g. create a context subscriptions). If the Dynamix Framework is deactivated, the application will receive the **`onSessionClosed`** event, which indicates that the application must wait until it receives **`onSessionOpened`** before calling additional IDynamixFacade methods. (Note that it is only necessary to call **`openSession`** once, even if an **`onSessionClosed`** event is raised, since Dynamix maintains the application's listener registration while deactivated.) Once the Dynamix Framework becomes active again, the **`onSessionOpened`** event will be raised and interaction with the Dynamix Service can continue.) If the application receives the **`onServiceDisconnected`** event[6] from the ServiceConnection object, Dynamix has probably been shut down by Android (or crashed). In this case, the application must call the IDynamixFacade's **`addDynamixListener`** method again, wait for **`onDynamixListenerAdded`**, then call **`openSession`** and wait for **`onSessionOpened`** before proceeding to call additional IDynamixFacade methods. Note that **`onSessionOpened`** always provides the application's current session id, which is used for some secure interactions with Dynamix, such as launching interactive context acquisition scans. The session id can also be found by calling the **`getSessionId`** method. The details of context scanning and context events are discussed in the next section.

[6] http://developer.android.com/reference/android/content/ServiceConnection.html

4. Context Scanning and Context Events

This section describes how applications can use the IDynamixFacade and IDynamixListener interfaces to manage context scanning and handle incoming context events.

4.1. Context Subscriptions

Perhaps the most important IDynamixFacade operations are related to managing context subscriptions. Briefly, a context subscription represents an active context modeling process that is performed by the Dynamix Service on behalf of a Dynamix listener. Context subscriptions are handled by one *or more* Context Plug-ins and are always created for a specific context type[7]. An application manages its context subscriptions using several IDynamixFacade methods (see section 3.1). Two important methods are shown below.

- **`addContextSubscription`**`(IDynamixListener listener, String contextType)`
- **`removeContextSubscription`**`(IDynamixListener listener, String contextType)`

When an application calls **`addContextSubscription`** using the IDynamixFacade, it is telling the Dynamix Service that a specific listener wishes to receive context events containing context information of the type specified by the **`contextType`** string. The **`contextType`** string specified *must* match a context type supported by one (or more) of the Context Plug-ins available in a given Dynamix Infrastructure. When **`addContextSubscription`** is called, the local Dynamix Service checks with its installed Context Plug-ins to determine if the requested context type can be supported. If the Dynamix Service is able to support the context subscription type (i.e. it has a suitable Context Plug-in installed), the Dynamix Service sets up the subscription, initiates context modeling, and raises the **`onContextSubscriptionAdded`** event. The **`onContextSubscriptionAdded`** event includes the original **`contextDataType`** string and the Context Plug-in that is handling the subscription. If multiple Context Plug-ins are capable of handling the subscription, separate **`onContextSubscriptionAdded`** events are raised for each Context Plug-in enlisted to support the subscription. If no Context Plug-in can be found to handle the context subscription type, the **`onContextTypeNotSupported`** event is raised. After context subscriptions have been added, they can be removed using the **`removeContextSubscription`** method by passing in the listener and context type of the subscription to remove. Several additional context subscription management methods are also available, as described in section 3.1.

In some scenarios, a Dynamix Service may not initially have support for a requested context subscription type, but will attempt to dynamically install an appropriate Context Plug-in to handle the requested context type. This process happens automatically if the requesting application has permission to install Context Plug-ins. During this process, the Dynamix Service first queries its Dynamix Infrastructure to see if a suitable Context Plug-in is available. If a suitable Context Plug-in can be found, a dynamic installation is performed and the **`onInstallingContextSupport`** event will be raised for each Context Plug-in being installed to handle the context subscription. The **`onInstallingContextSupport`** event includes the Context Plug-in being installed in addition to the context subscription type being supported. If a Context Plug-in is

[7] For an up-to-date listing of supported context types, see: TODOs.

successfully installed for a context subscription request, the **`onContextSubscriptionAdded`** event is raised. If the Context Plug-in cannot be installed, the **`onContextTypeNotSupported`** event is raised.

Related, applications can also request that the Dynamix Service install a specific Context Plug-in using the **`requestPluginInstallation`** method. This method takes a fully qualified, string-based URL of a Context Plug-in Description Document (see TODO). If the application has permission to install Context Plug-ins, the Dynamix Service will attempt to install the plug-in using the specified URL. At the beginning of the installation process, the **`onInstallingPlugin`** event is raised. If the Context Plug-in is successfully installed, the **`onPluginInstalled`** event is raised. If the Context Plug-in installation fails, the **`onPluginInstallfailed`** event is raised. Once a Context Plug-in is installed using this method, context subscriptions must still be made using the **`addContextSubscription`** method, as described above.

4.2. Managing Context Modeling

If a context subscription is successfully added, the Dynamix Service will begin modeling the specified context type using a specific Context Plug-in (or set of plug-ins). Depending on the type of Context Plug-in involved, applications may need to interact with the Context Plug-in to manage parts (or all) of the context modeling process. The Context Plug-in type can be determined by calling the **`getContextPluginType`** method of the **`ContextPluginInformation`** object provided by the **`onContextSubscriptionAdded`** event. There are several basic types of Context Plug-ins, which are summarized in Table 4.

Context Plug-in Type	Description
ContextPluginType.PUSH No application interaction required	Performs *continuous context modeling* for a specific set of context info types, while broadcasting context events to all Dynamix listeners holding an associated context subscription (and appropriate security credentials). These plug-ins "push" context events to clients without requiring applications to specifically request a context scan.
ContextPluginType.PULL Application interaction required	Performs *single* context modeling scans in response to a Dynamix listener's requests to do so. Unlike push Context Plug-ins, they do not operate independently in the background.
ContextPluginType. PULL_INTERACTIVE Application interaction required	Same as ContextPluginType.PULL, but requires user interaction via an interface provided by the Context Plug-in. Applications launch an interactive pull context modeling scan using standard Android Intents, as described in section 4.2.2.
ContextPluginType.PUSHPULL Some application interaction may be required	Combines both the push and pull functionality described above. As with the a push Context Plug-in, PushPull Context Plug-ins "push" context events to clients without requiring listeners to specifically request a context scan. In addition, like the pull Context Plug-in, PushPull Context Plug-ins implementations also "pull" context information from the environment in response to an application's requests to do so.
ContextPluginType. PUSHPULL_INTERACTIVE Some application interaction may be required	Same as ContextPluginType.PUSHPULL, but pull functionality requires user interaction via an interface provided by the Context Plug-in.

Table 4: Context Plug-in Types

4.2.1. Push Context Plug-ins

Context Plug-ins of type ContextPluginType.PUSH require no application interaction to model context. Once a context subscription is created using a push Context Plug-in, the plug-in operates independently in the background, modeling context information and broadcasting context events as needed. Modeled context information is sent to the application using the context event model described in section 4.3.

4.2.2. Pull Context Plug-ins

Unlike push Context Plug-ins, pull Context Plug-ins *require* application interaction when modeling context information. As shown Table 4, there are two variants of the pull Context Plug-in type: ContextPluginType.PULL and ContextPluginType.PULL_INTERACTIVE. When using a Context Plug-in of type ContextPluginType.PULL, listeners trigger a context modeling scan using the **`requestContextScan`**`(String pluginId, String contextType)` method of the IDynamixFacade. When calling the **`requestContextScan`** method, the listener provides the pluginId of the Context Plug-in to use, plus a string describing the context type the application wishes to receive. (The pluginId can be found using the **`getPluginId()`** method of the `ContextPluginInformation` object provided by the **`onContextSubscriptionAdded`** event or from the list of Context Plug-in Information returned from the IDynamixFacade's **`getContextPluginInformation`** method.) Related, the **`requestConfiguredContextScan`** method allows the listener to provide a context scan configuration in the form of an Android Bundle. The configuration options for each plug-in (if supported) are specified by the Context Plug-in developer. The **`requestContextScan`** and **`requestConfiguredContextScan`** methods operate asynchronously, delivering its results using the context event model described in section 4.3.

When using a Context Plug-in of type ContextPluginType.PULL_INTERACTIVE, applications start the Context Plug-in's context acquisition interface using an Android Intent, which allows users to manually control the context modeling process with a user interface provided by the Context Plug-in (e.g. point a camera, input data, etc.) To start a Context Plug-in's context acquisition user interface, the listener creates an Android Intent using an ACQUIRE_CONTEXT action type, as shown in Figure 2. The intent must include the target Context Plug-in's plug-in identifier, the application's session id and the listener's listener id (as extended Intent data), using the keys "pluginId", "sessionId" and "listenerId" respectively. Some plug-ins may also support configuration options using additional extended Intent data (see the specific Context Plug-in's documentation for details). As described in Table 2, the application's session id can be obtained through the IDynamixFacade's **`getSessionId`** method and the listener id can be obtained using the **`getListenerId`** method. If the application has permission to launch the interface, the Context Plug-in's user interface appears in the application's Activity stack, as per the Android application model. The user then controls the context modeling process using the interface, which may result in context information being sent to the application using the context event model described in section 4.3.

```
Intent intent=new Intent("org.ambientdynamix.contextplugin.ACQUIRE_CONTEXT");
intent.putExtra("pluginId", "org.ambientdynamix.contextplugins.barcode");
intent.putExtra("sessionId", dynamixInterface.getSessionId());
intent.putExtra("listenerId", dynamixInterface.getListenerId(callback));
startActivity(intent);
```

Figure 2: Launching a Context Plug-in's Context Acquisition User Interface From an Android Activity

4.2.3. PushPull Context Plug-ins

As described in Table 4, a PushPull Context Plug-in combines *both* push and pull functionality. Foundationally, a PushPull plug-in operates independently in the background, modeling context information and broadcasting context events, as described in section 4.2.1. In addition, a PushPull plug-in simultaneously listens for context scan requests, as described in section 4.2.2. Both types of pull-based context acquisition modes are supported: a plug-in of type ContextPluginType.PUSHPULL listens for **`requestContextScan`** method calls; and a ContextPluginType.PUSHPULL_INTERACTIVE plug-in listens for ACQUIRE_CONTEXT Intents and launches a user-driven context acquisition user interface in response. In all modes, results from the context modeling processes are delivered to applications using the context event model described in section 4.3.

4.3. Context Events

After a listener creates a context subscription, as described in section 4.1, and manages the context modeling process, as described in section 4.2, discovered context information will be sent from the Dynamix Service to the listener as context events (see the **`onContextEvent`** method introduced in Table 3). The key methods of the ContextEvent object are shown in Table 5.

Method Summary	
`String`	**`getContextType`**`()` Returns the type of the context information represented by the event.
`ContextPlugin Information`	**`getEventSource`**`()` Returns the ContextPluginInformation for the plugin that generated the event
`IContextInfo`	**`getIContextInfo`**`()` Returns the event's IContextInfo (if present).
`String`	**`getStringRepresentation`**`(String format)` Returns a string-based representation of the event's IContextInfo for the specified format string (e.g. "application/json") or null if the requested format is not supported.
`Set<String>`	**`getStringRepresentationFormats`**`()` Returns a Set of supported string-based context representation format types or null if no representation formats are supported.
`boolean`	**`hasIContextInfo`**`()` Returns true if this event has IContextInfo; false otherwise.

Table 5: ContextEvent Key Methods

Note that a context event may support *multiple* string-based formats (e.g. both "application/json" and “text/xml”). The available string-based formats are provided in the Context Plug-in’s documentation.

Additionally, 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 6.

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 6: Expirable Key Methods

4.3.1. The IContextInfo Interface

Each Context Plug-in may provide one or more IContextInfo implementations, which provide native (i.e. object-based rather than string-based) contextual information appropriate to the context domain. The IContextInfo interface enables a Dynamix Service to support a broad range of context types not known at compile time. To enable applications to work with native context objects, Context Plug-in developers may release JAR files containing their associated data types, which can be included in the application's build path. If a Dynamix application does not have the classes necessary to parse the IContextInfo entity in the event, the context event's **getEventInfo** method will return null. In this case, the Dynamix application may use a string-based representation of the context information, which is returned by the context event's **getStringRepresentation** method (see Table 5). The available representation formats can be found by using the context event's **getStringRepresentationFormats** method (see Table 5). If the application has the required IContextInfo classes, the context event's **getEventInfo** method will return the IContextInfo object. The IContextInfo object's data type is self describing, as shown in Table 7.

IContextInfo: Key Methods	
String	**getContextType()** Returns the type of the context information represented by the IContextInfo. This string must match one of the context type strings described by the source ContextPlugin.
String	**getImplementingClassname** () Returns the fully qualified class-name of the class implementing the IContextInfo interface. This allows Dynamix applications to dynamically cast IContextInfo objects to the original type.

Table 7: IContextInfo Key Methods

If IContextInfo is present within an event, the listener has two primary ways of casting the object to the proper type. First, the listener can use the **`getImplementingClassname`** method of the IContextInfo interface, which returns the fully qualified class-name of the class implementing the IContextInfo interface. Second, the listener can use a Java `instanceof` comparison to check for proper types. An example of dynamic type casting is shown in Figure 3.

```
public void onContextEvent(ContextEvent event) throws RemoteException {
  // Check if the event has IContextInfo
  if(event.hasIContextInfo()){

    // Check for IContextInfo of type SamplePluginContextinfo
    if(event.getIContextInfo() instanceof SamplePluginContextinfo)){

      // Now we can safely cast the IContextInfo into a SamplePluginContextinfo
      SamplePluginContextinfo eventInfo = (SamplePluginContextinfo)
           event.getIContextInfo();

      // In this simple example, just print out the sample data
      System.out.println(eventInfo.getSampleData());
     }
   }
```

Figure 3: Dynamically Casting an IContextInfo Object to its Proper Type