PreEmptive Solutions

PreEmptive Analytics Development Tips and Recommendations

The fine art of PreEmptive Analytics

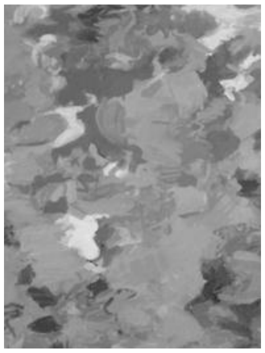
Only the developer knows what runtime data can provide critical insight into software quality, application value and user experience.

Through a unique set of technologies that capture, transport, analyze and publish custom, complex and rich data, PreEmptive Analytics has taken a fundamentally different approach to application monitoring that sets it apart from web analytics, network monitoring and debugging alternatives; *but with that power comes responsibility.*

In order to guarantee that developers have the freedom to gather what they require without impacting application performance or stability itself, PreEmptive Analytics implementations are configured entirely by the developer.

Like a painter presented with a blank canvas, the developer paints a picture of their app by adding just the right details but no more.

Whereas operations monitoring solutions deliver a mountain of data forcing the developer to chip away like a sculptor and web tools restrict the developer's palette to the simplest of colors, PreEmptive Analytics offers development a complete spectrum of options—from system to session to application and user-specific data.


The blank canvas of PreEmptive Analytics offers development freedom but holds special rewards for good design.

Whether developing for the enterprise or the consumer, the cloud or the data center, the web or the phone, the following articles offer specific techniques to satisfy both users and stakeholders alike.

Oops—did I do that? (*Analytics gone wrong*)

The following examples are all true: the app and the developers' identities have been hidden to protect the innocent (or at least the naïve).

Feature usage does not equal network monitoring. Feature tracking was placed on the "packet received" method inside a popular open source network device driver.

Hardware specs are not user preferences. A multi-Mb manufacturing spec was stuffed into a single custom data property.

Runtime is not real-time. A performance probe was placed inside a mobile game's "physics engine."

If these examples make you cringe (and especially if they don't), the following articles should serve you well as a development resource.

Special points of interest:

- *Tips to monitor apps with especially heavy usage*
- *Tips to capture actionable exceptions versus every ops hiccup*
- *Capture custom data that matters*
- *Avoid common programming mistakes*

Inside this issue:

MINIMIZING MESSAGE VOLUME[1]

Message volume should be kept to a reasonable level commensurate with the performance, bandwidth and costs associated with the application and platform.

⇒ Avoid putting attributes on frequently executed methods (e.g. update/draw methods in games, UI events that trigger frequently, etc.).

This prevents excessive data collection and unnecessary application performance penalties.

⇒ Try to collect data in response to specific user actions, occasionally occurring events, or exceptional situations.

⇒ For heavily used features that must be measured, consider the techniques outlined in "OPTIMIZING MESSAGE VOLUME."

As a general principle, it is a good practice to avoid sending unnecessary messages.

THE ON SWITCH: STARTING UP

Setups and Teardowns should completely cover entry and exit points.

⇒ Ensure there is no way to start the application without hitting a SetupAttribute. For example, you will want to tag both the Application_Started and Application_Launching methods in WP7 apps.

⇒It is possible that one might want to only instrument a specific part of an application that is guaranteed to execute no more than once per app run – in this case, be sure to completely cover all entry and exit points for that specific part of the application.

⇒ Make sure entry points have the same CustomEndpoint and UseSSL option.

⇒ Configure an OptInSource if you want to give the user an opportunity to opt in/out of message collection. Once this Boolean value is read during the injected code for the SetupAttribute, it cannot be changed until the next app run.

⇒ Configure an InstanceIdSource if you want to be able to uniquely identify each individual user. You can use any runtime string (e.g. the user's email address, serial number…) or you can generate and store a GUID.

The Serial Number report shows data by serial number, and the filter bar allows you to filter by the InstanceId / Serial Number.

⇒ Configure an ExtendedKeySource to collect any custom data to be reported once at the beginning of each app run. For example, you might collect information about available devices and peripherals relevant to your domain that are unlikely to change during the application run.

THE OFF SWITCH: CLEANING UP

App monitoring is deactivated with the TeardownAttribute. It flushes the cache of any queued data and other "house keeping" functions. Here are some tips to consider:

⇒ Cover all exit points.

⇒ Configure an ExtendedKeySource to collect any custom data to be reported once at the end of each app run. For example, you might collect information about the peak memory usage of the app or the number of times certain tasks were executed.

Note, this latter example is covered in more detail in the discussion on FeatureAttributes but the short explanation is that sometimes it is impractical to report every single instance of a feature and so it is more efficient to count of the number of executions over an app run as an alternative.

[1] Please refer to the appropriate product documentation for additional information on all referenced attributes and software features.

Optimizing Message Volume

It is a good practice to avoid sending unnecessary messages. The following techniques have proven effective in scaling analytics with adoption.

⇒ Do not use FeatureAttributes to track a feature (or method) that is executed in high volumes. An alternative is to create a counter that is incremented each time the feature is used. The counter value can be sent as ExtendedKey data on the TeardownAttribute. You will still know how many times the feature has been used while minimizing bandwidth and processing power utilization. One potential issue arises when the TeardownAttribute does not occur for a session. If this is a concern, you can add the same count to a FeatureAttribute that you expect to happen only occasionally and then zero out the counter each time it's reported.

⇒ Configure an ExtendedKeySource to prevent redundancy between similar features. For example, you may have an application with 50 pages the user can visit. Instead of having 50 features named "Overview Page," "About Us Page," "Contact Page," etc., you can create just one feature called "Page Viewed" with associated custom data where the Key is always "Page Name" and the Value is the name of the current page. This would also work for tracking shopping cart progress and other similar tasks.

These may be on the same or different methods.

Exception Monitoring

Exception monitoring often requires special consideration in that this is where development and operations often first overlap. These are, by their very definition, event driven. The following techniques will help to separate meaning full events from the rest of the noise.

⇒ Identify what type of exceptions that should be tracked. There are four main categories of exceptions:

Assembly-level Unhandled – track exceptions that propagate out of the user code and crash the application. This is a great starting point because most everyone wants to know about exceptions that crash their application.

Method-level Unhandled – tracks exceptions that leave the target method (regardless of whether it goes on to crash the application). This is useful if you have an entry point method to a large feature that should, in theory, not cause exceptions. Adding a method-level unhandled ExceptionTrackAttribute to that method will report cases when exceptions leave that method.

Caught – track exceptions that are caught by "catch" blocks. These can be added to individual methods or the assembly (which is equivalent to adding it to each individual method).

Thrown – track exceptions that are thrown by "throw" instructions. These can be added to individual methods or the assembly (which is equivalent to adding it to each individual method).

⇒ Configure a ReportInfoSourceElement to surface an exception report opt-in (and option to provide contact and context info) to the user. If your app runs on the desktop or on Silverlight in the browser, then you can use the DefaultAction ReportInfoSourceElement which will show a default opt-in dialog with contact and context info input boxes. If you use the DefaultAction ReportInfoSourceElement, then you can specify a PrivacyPolicyUri that is surfaced as a link in the default opt-in dialog.

⇒ Configure an ExtendedKeySource to collect any custom data to be reported once per exception incident. For example, you might collect information about the environment that may be volatile during application execution to find a trend between user environments where the exceptions are happening.

For heavily used features that must be measured, consider the techniques outlined in "Optimizing Message Volume."

INTERESTED IN LEARNING MORE?

CONTACT US AT

+1 440-443-7200

OR EMAIL SOLUTIONS@PREEMPTIVE.COM

CUSTOM DATA: EXTENDED KEYS

Often, to best understand application impact and quality, it is helpful to be able to collect and analyze custom data unique to an application, a user, a particular session or system. While PreEmptive Analytics can handle most every type of data, the following provides an overview of some restrictions and recommendations.

There are some reasonable data limitations.

- Keys must be ≤ 2000 Unicode characters
- Values must be ≤ 4000 Unicode characters
- Extended keys are reported as either String or Numeric values.

String valued extended keys display every individual Value reported for the Key, along with a % of total. This makes it useful when there's a reasonably sized set of possible Values and you want to see each Value and how many times it appeared.

Numeric valued extended keys display each Key only once, with a count, average, min, and max. This makes it useful for numeric data on which one would want these types of metrics instead of a full list of values.

⇒ Only use ToString() on objects if you are certain it formats the data in a useful way. You may need to write utility methods to format the data contained by specific types of objects to make the data visible and useful inside the portal.

TO BEST UNDERSTAND APPLICATION IMPACT AND QUALITY, IT IS HELPFUL TO BE ABLE TO COLLECT AND ANALYZE CUSTOM DATA UNIQUE TO AN APPLICATION, A USER, A PARTICULAR SESSION OR SYSTEM.

SYSTEM AND PERFORMANCE PROFILING

In addition to usage monitoring and application events, it's also often useful to track system and runtime stacks and resource utilization. The following techniques help to ensure efficient system and performance monitoring.

⇒ To track the cost of a specific process or algorithm that in terms of memory or CPU usage, use the PerformanceProbeAttribute. Use a PerformanceProbeAttribute once at the beginning of the process (be sure to specify a descriptive Name that indicates it's at the beginning of the process) and another PerformanceProbeAttribute at the end of the process (with a similarly descriptive name). These can be on the same or different methods – use the InjectionPoint of Beginning/End to specify at what point in the method to take measurements.

⇒ Use PerformanceProbeAttributes infrequently as it requires time to gather the appropriate information.

⇒ Similarly, use the SystemProfileAttribute infrequently as it also requires time to gather the appropriate information.