DateTime Issues for Rock

# Problem

Rock will often be hosted in a time zone that is different that the Church that is using Rock. For example, Rock might be hosted on a server in New York (Eastern Time Zone, with Daylight Savings Time) but is being used by a church in Arizona (AZ Time, without Daylight Savings Time). If the application does not account for this, the date/time values that are stored in the database will be off by 2 hours for part of the year and 3 hours for the other part of the year. Also, Date/Time comparisons like “How long until the 10am class opens?” will be incorrect. There are also other problems, such as “this workflow needs to process records every night at 2:30am”. This process would run twice on Daylight Savings Fallback day, and get skipped on Daylight Savings Skip-ahead day.

# Uses of Date and time information in an application

From <http://msdn.microsoft.com/en-us/library/bb384267.aspx>, the more common uses of date and time information include one or more of the following:

* To reflect a date only, so that time information is not important.
* To reflect a time only, so that date information is not important.
* To reflect an abstract date and time that is not tied to a specific time and place (for example, most stores in an international chain open on weekdays at 9:00 A.M.).
* To retrieve date and time information from sources outside the .NET Framework, typically where date and time information is stored in a simple data type.
* To uniquely and unambiguously identify a single point in time. Some applications require that a date and time be unambiguous only on the host system; others require that it be unambiguous across systems (that is, a date serialized on one system can be meaningfully deserialized and used on another system anywhere in the world).
* To preserve multiple related times (such as the requestor's local time and the server's time of receipt for a Web request).
* To perform date and time arithmetic, possibly with a result that uniquely and unambiguously identifies a single point in time.

# Definitions

* DateTime
  + The SQL Server datatype that stores Date and Time, but without any time zone information.
  + The .NET structure that is used to manipulate Date and Time data. It represents a point in time, but without any time zone information.
* DateTimeOffset
  + A SQL Server datatype (introduced in SQL Server 2008) that stores Date and Time, and also stores the offset of this Date and Time relative to UTC.
  + The .NET structure that is used to manipulate Date and Time data. It represents a point in time relative to UTC.

# How to solve the Problem

* First of all, why is this is such a problem in Rock? Haven’t we had other ChMS systems run successfully without having to worry about this?
  + Up until now, most ChMS systems are hosted in the same time zone as the Church that it serves, often in the same building. In these cases (as long as the clocks are reasonable synced across the users computers), time zone problems are not an issue. The Daylight Savings Time problem still is, but at least the server and church have the same Daylight Savings Time rule, so the problem is fairly minor (and it is a relatively familiar problem that people have dealt with before)

# Minimum Solution

* At a minimum, the Rock application will have to be designed so that the problem is no worse in hosted environments compared to systems that are hosted on site. So, the solution would have to somehow emulate that it is in the same time zone as the church. This would require that any code that uses DateTime functions to factor in the Application-Defined TimeZone instead of the Server TimeZone.
* Requirements of Minimum Solution
  + Need an Application-Defined Timezone to be configured. For example, a web.config setting of ApplicationTimeZone=<TimeZoneName>, where the TimeZoneName would represent the church’s timezone. The TimeZoneName choice would come from a List provided from the Operating System, in Rock’s case, a Windows Server (another option is the Olsen TimeZone list which is a Unix standard)
  + A helper class (ex: RockDateTime) that would need to be used instead of the normal .NET DateTime class.
* Advantages of Minimum solution
  + Assuming that absolutely every DateTime function uses the RockDateTime helper class, the problem is no worse than existing onsite hosted ChMS systems
* Disadvantages of Minimum solution
  + If the Application-Defined Timezone is misconfigured or changed during the course of system’s lifetime, any recorded date/time data will be incorrect
  + Any date/time data that is stored as a result of using DateTime instead of RockDateTime will be incorrect. Also, it would be very easy to not notice this happening. Assuming the bug is eventually discovered, it would be difficult to fix up any incorrect date/time data that resulted from it.
  + Date/Time calculations that use DateTime instead of RockDateTime would cause incorrect results. For example, if a 10:00am class has a workflow or process that needs to start at 9:45am, this process/workflow could happen at the wrong time (for example, 2 hours early, or an hour too late)
  + Developers will easily miss these kinds of bugs since they typically test their software in the same timezone as the target environment.
* Proposed SQL DataTypes and Naming Standards for Minimum Solution

|  |  |  |  |
| --- | --- | --- | --- |
| **Usage** | **.NET Type** | **SQL DataType** | **Naming Suffix** |
| Date | DateTime | Date | …Date |
| Time | TimeSpan | Time | …Time |
| DateTime | DateTime | DateTime | …DateTime |

# Alternate Solution

* For Date usages and Time usages, use the same solution as the Minimum Solution, but utilize the DateTimeOffset class and datatype for DateTime usages. There would still be a need for an Application-Defined TimeZone, but an incorrect or changed setting would not result in incorrectly stored date/time data. In short, it would shift the problem to a rendering issue instead of a storage issue.
* Requirements of Alternate Solution
  + Need an Application-Defined TimeZone (same as Minimum Solution)
  + Does not need to use a RockDateTime helper for DateTime storage. DateTime comparison logic would have to be aware of the Application-Defined timezone, but only in certain situations.
  + Displayed Date/Time values would need to factor in Application-Defined TimeZone
* Advantages of Alternate Solution
  + The Application-Defined TimeZone doesn’t need to be factored in for DateTime storage (it is a little better if it does, but not mandatory)
  + Developers can use the built-in .NET DateTimeOffset class instead of requiring them to use a special RockDateTime class. Depending on what they are doing, they should use the DateTimeOffset class with the Application-Defined timezone, but if they don’t, most things will still work, and there will never be a date/time storage bug due to not using the RockDateTime helper. Even if they just use .NET DateTime instead of DateTimeOffset, most things will work properly since DateTime to DateTimeOffset storage automatically records the proper point-in-time and offset.
  + Stored DateTime’s will accurately represent the intended point in time.
  + Applications that use the server timezone instead of the application-defined timezone for storing data will be detectable by looking at the data.
  + Daylight Savings Time is much less of an issue, and won’t result in ambiguous date/time data.
* Disadvantages of Alternate Solution
  + A person writing direct SQL against a DateTimeOffset column could get incorrect results. We had originally considered this a deal-breaker. However, using a column naming standard of a DateTimeOffset suffix, along with a Computed DateTime column would nearly eliminate this problem.
  + Every DateTimeOffset column would need to have an associated computed DateTime column.
  + Developers would need to understand how DateTimeOffset works, especially if they are doing date/time comparison logic.
  + There are more problems with DateTime vs DateTimeOffset, but the problems are more familiar to developers.
* Proposed SQL DataTypes and Naming Standards for Alternate Solution

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Usage** | **.NET Type** | **SQL DataType** | **Naming Suffix** | **Note** |
| Date | DateTime | Date | …Date |  |
| Time | TimeSpan | Time | …Time |  |
| DateTime | DateTimeOffset | DateTimeOffset | …DateTimeOffset | Include Computed DateTime column |

# Issues with either solution

* Working with external systems (like a payment processor) would require a conscious effort when dealing with DateTime. This effort might be a bigger issue in one solution versus the other, depending on the problem and the skillset of the developer.

# Advantages of Alternate Solution outside of the current Rock Requirements

* Rock "could" be extended at some point in the future to have multiple timezone support without having to update previously stored datetimes.
* Rendered Date/Time values could include information about which TimeZone offset the data was originally recorded in.