ChronoZoom® beta

Developer's Guide

ChronoZoom® beta

Contents

Disclaimer

Introduction

ChronoZoom is an open source community project dedicated to visualizing Big History that has been funded and supported by Microsoft Research Connections in collaboration with University California at Berkeley and Moscow State University.

You can browse through history on ChronoZoom to find data in the form of articles, images, video, sound, and other multimedia. ChronoZoom links a wealth of information from five major regimes that unifies all historical knowledge collectively known as Big History.

This guide is designed to walk developers through the installation process. Some of the APIs for ChronoZoom are not yet available for the public.

Prerequisites

Hardware Requirements:

- A computer that can run Visual Studio 2010.
- Optionally, a network connection for the content links.

Software Requirements:

- Visual Audio 2010 with Service Pack 1
- Installation of Azure SDK
    - Recommend installing from this page: http://www.windowsazure.com/en-us/develop/downloads/
- SQL Server Developer Tools

- SQL Server Standard, Express, Developer, or Enterprise Edition

Project setup instructions

Here are the steps to get the source code and prepare for execution:

1. Download the code as zip file from Codeplex at http://chronozoom.codeplex.com/
2. Extract the zip into a folder.
3. Open the chronozoom.sln under the source folder.
4. Right-click the Chronozoom.Database in Visual Studio Solution Explorer and click **Publish.**
5. In the dialog box, select the target database connection string to deploy to.
6. Click Publish.
7. In the chronozoom.UI project, under the scripts folder, locate the cz.settings.js
    a. Change the text '[Your blob name]' to your blob's name.
8. Replace the text [Your domain] in the chronozoom.svc.cs file with your domain address
9. Replace the text [Your domain] in the ChronozoomRelay.cs file with your domain address
10. Set ChronoZoom.UI as the startup project.
11. Run and execute.

Coding Guidelines

One of the key components to delivering that higher productivity is by providing a consistent approach to the programming model and stylistic conventions used throughout the development of the project. Consistency reduces distractions and surprises.

Our intent is to follow the guidelines laid out by the designers of the .NET Framework class library as detailed in the book *"Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries"* by Krzysztof Cwalina and Brad Abrams, published by Addison-Wesley in 2005 and described in the MSDN documentation delivered with Visual Studio (found online here). While we principally follow the guidelines above, conversations and styles from many other projects and developers have had influence on this document.

Coding Conventions

Document Conventions

Just as writing code in a consistent style makes the code easier to read, this document uses words and formatting to ensure clarity and comprehension. The conventions used herein are recommendations and should be followed in most situations; however, we recognized that there are situations we may not be able to anticipate. Thus our recommendations are stated with a 'strength' context verb to help you decide how to apply them in your situation. Our terminology is consistent with the Framework Design Guidelines.

Our strongest recommendations are **DO** and **DO NOT**. **DO** and **DO NOT** recommendations should be followed 99.999% of the time to obtain the benefits of a consistent coding style. For example:

> **DO** use descriptive parameter names.
> **DO NOT** require user to explicitly instantiate more than one type in the most basic scenarios.

Our general recommendations are **CONSIDER** and **AVOID**. These guidelines should generally be followed, but we are aware of circumstances where they may not make sense. If you understand the reasoning for the general recommendation and you fully understand your situation and the reasons you need to violate the recommendation are clear, then go ahead. For example:

> **CONSIDER** defining interfaces to achieve a similar effect to that of multiple inheritances.
> **AVOID** using out or ref parameters.

Naming Conventions

Identifier	Public	Protected	Internal	Private	Notes
Project File	PascalCase	-	-	-	DO match Assembly and Namespace.
Source File	PascalCase	-	-	-	DO match Class name.
Namespace	PascalCase	-	-	-	DO use Project/Assembly name.
Class or Struct	PascalCase	PascalCase	PascalCase	PascalCase	CONSIDER adding suffix of sub-class.
Interface	PascalCase	PascalCase	PascalCase	PascalCase	DO prefix Interfaces with capital I.
Generic Class	PascalCase	PascalCase	PascalCase	PascalCase	DO use T or K as type identifier.
Method	PascalCase	PascalCase	PascalCase	PascalCase	CONSIDER using Verb or Verb-Object pair.
Property	PascalCase	PascalCase	PascalCase	PascalCase	DO NOT prefix with Get or Set.
Field	PascalCase	PascalCase	PascalCase	_camelCase	AVOID use of public Fields.
Static Field	PascalCase	PascalCase	PascalCase	_camelCase	
Constant	PascalCase	PascalCase	PascalCase	_camelCase	
Enum	PascalCase	PascalCase	PascalCase	PascalCase	
Delegate	PascalCase	PascalCase	PascalCase	PascalCase	
Event	PascalCase	PascalCase	PascalCase	PascalCase	
Local variable	-	-	-	camelCase	
Parameter	-	-	-	camelCase	

Coding Style and Language Usage Conventions

Coding Style Issue	Style Guideline
Source Files	DO have only one Namespace and one Class per file.
Curly Braces	DO have braces on a new line. DO use a brace, even if syntactically optional.

Coding Style Issue	Style Guideline
Indentation	Do indent is 4 and convert tabs to spaces.
Comments	Do use // or ///.
Native Data Types	DO use built-in C# data types.
Enums	AVOID changing the default type.
Generics	CONSIDER generic types before standard or strong-typed classes.
Methods	AVOID large parameter lists (over 7). Use an array or object instead.
foreach	DO NOT modify enumerated objects within the body of the foreach loop.
Conditionals	DO NOT test against true or false. DO NOT embed code with side effects within a conditional. DO NOT embed an assignment. DO NOT embed a method invocation.
Exceptions	DO NOT use exceptions for normal flow control. DO NOT use "throw e;" when re-throwing, use "throw;". DO avoid exceptions by validating pre/post conditions prior to the exception. DO derive from Exception, not ApplicationException.

Naming Conventions

Consistent naming procedures applied across all the various identifier types in a project goes a long way to making code more accessible to people that use the library or join the project after you.

The CLR supports both case-sensitive and case-insensitive languages. If you want to write libraries that are accessible to both types, you are restricted from on creating names that are identical except for case. Even so, appropriate and consistent capitalization enhances readability and comprehension of code.

Capitalization is used to make identifiers more readable. There are two capitalization conventions in general use for identifier names throughout the library, PascalCase and camelCase. PascalCase capitalizes every word of the descriptive identifier whereas the camelCase does not capitalize the initial word of the identifier but does capitalize every word after the initial one. There are some instances where a PascalCase or camelCase identifier may have a '_' or a capital 'I' prefix added.

General Naming Guidelines

1. DO use PascalCasing for type and method names and constants.
2. DO use camelCasing for local variable names and method arguments.
3. DO use 'I' to prefix Interface names.
4. DO NOT create names that vary only by case.
5. AVOID creating ALLCAPITALNAMES.
6. DO use easily readable, descriptive, meaningful, and specific names.
    a. AVOID single character names.
    b. AVOID Hungarian notation for public or protected members.
    c. AVOID abbreviations unless the full name is really excessive.
    d. AVOID adding redundant prefixes and suffixes e.g. public enum ColorsEnum {…}.
7. DO name types with nouns, noun phrases, or adjective phrases using PascalCasing.

8. CONSIDER ending the name of a derived classes with the name of the base class.
9. DO prefix Interface names with the letter 'I', to indicate that the type is an interface.
10. DO give methods names that are verbs or verb phrases.
11. DO name properties using a noun, noun phrase, or adjective.
12. CONSIDER giving a property the same name as its type.
13. DO name fields using a noun or noun phrase.
14. CONSIDER using common, well known acronyms, e.g. IO instead of InputOutput.
15. AVOID acronyms that are not common or widely known.
16. DO use uppercase for two-letter acronyms and PascalCase for other acronyms. It is acceptable to use "BIO" rather than "Bio" in the context of this project.
17. CONSIDER using C# types rather than aliases in the System namespace.
18. DO use capital letters for type placeholders in generics.
19. CONSIDER prefixing Boolean variables with "Can", "Is", or "Has".
20. CONSIDER appending computational variables with the suffix "Average", "Count", "Sum", "Min", or "Max".
21. DO use meaningful namespaces under the "Bio" root when extending the Framework.
22. DO use meaningful namespaces under your Product, Company, or Developer name for tools or code using the project.e.g. namespace MyCompany.MyProduct.MyNamespace;

Coding Style and Language Usage

Resolving issues around different coding styles can cause controversy among developers, but consistent layout, format, and organization are key attributes of maintainable code. The following guidelines describe the 'preferred' way to write C# code in order to consistently create clear, readable, comprehensible code that you and your team members can maintain over time.

General Style Guidelines

1. DO have a single file contribute to a single namespace.
2. DO have a single class in a single source file.
3. DO group all framework "using" namespaces together before custom or third party namespaces.
4. AVOID putting a "using" statement inside a namespace.
5. AVOID using fully qualified type names by using the "using" statement.
6. CONSIDER grouping internal class implementation by type in the following order:
    a. Member variables.
    b. Constructors & Finalizers.
    c. Nested enums, structs, and classes.
    d. Properties
    e. Methods
7. DO explicitly declare the access modifier for all identifiers rather than relying on default.
8. CONSIDER grouping declarations within each type by their access modifier and visibility:
    a. public
    b. protected
    c. internal
    d. private

9. DO declare all member variables as private and provide public, protected, or internal property access where required.
10. DO declare each variable in its own statement.
11. DO use white space to organize and separate code.
12. DO maintain strict indentation of 4 spaces. DO NOT use tab characters in the documents.
13. DO indent all code blocks contained within curly braces.
14. DO place opening curly braces ({) on a new line.
15. DO indent comments to the same level as the code being described.
16. DO use correct spelling, grammar, and punctuation in all comments.
17. DO use // or /// for comments.
18. AVOID /* … */ for comments.
19. DO use inline comments to explain assumptions, issues, and algorithmic insights.
20. DO NOT use comments to explain obvious code.
21. DO use Task-List keywords in comments to allow filtering e.g.
    ```
    // TODO:
    // UNDONE:
    ```
22. DO use C# comment blocks for documenting the API.
23. DO use C# comment blocks for every public, protected, and internal declaration.
24. DO include <summary> comments.
25. DO include <param>, <return>, and <exception> comments where applicable.
26. DO declare local variables near their first use.
27. DO initialize local variables where you declare them.
28. DO use the simplest data type, collection, or object that meets your requirements.
29. AVOID specifying the type of an Enum. Use the default of int unless you have an explicit need for a long.
30. DO NOT use inline numeric literals (‘magic numbers’). Use Constant or Enum.
31. AVOID declaring string literals. Use Resources, Constants, Configuration Files, Registry or other data sources.

Library Design

A successful framework for biology must be designed for a broad range of developer skills and capabilities. Delivering powerful capabilities with clean, simple abstractions that can be used by the novice, while still allowing the expert to control the details underlying the abstraction is a challenging task.

General Library Design Guidelines

1. DO design a framework that is both powerful and easy to use.
2. DO provide a layered framework with high-level APIs optimized for productivity and low-level APIs optimized for power and expressiveness.
3. AVOID mixing low-level and high-level APIs in a single namespace.
4. DO ensure that layers of a single feature are well integrated and complete. Developers should be able to migrate from one level to another without re-writing the entire application.
5. DO base your design on meaningful usage scenarios.
6. DO ensure scenarios that correspond to an appropriate abstraction level.

7. DO design APIs by first writing code samples for the main scenario then defining the object model to support the code.
8. DO organize usability studies to test API in main scenarios.
9. DO provide simple overloads of constructors and methods with a small number of primitive parameters.
10. DO NOT require user to explicitly instantiate more than one type in the most basic scenarios.
11. DO provide good defaults for all properties and parameters if possible.
12. DO communicate incorrect usage of APIs using exceptions.
13. DO provide strongly typed APIs if at all possible.
14. DO NOT create circular references between assemblies.
15. CONSIDER factoring unsafe code into a separate assembly.
16. DO use zero-based indexing when developing with the Framework.

Azure Deployment Guidelines

ChronoZoom is powered by Windows Azure and SQL Azure. With Team Foundation Server as the source repository for ChronoZoom, there comes an added benefit of build servers. We leverage the build feature of TFS to automatically deploy packages to Azure.

As part of the Engineering practice, it would be ideal to let TFS handle the deployment to Azure as part of the daily builds. Here are some advantages of doing this:

- Being able to build, package, and deploy our projects to multiple Windows Azure environments from a build server using MSBuild
- Being able to retain copies of our packages and configurations for our builds in Windows Azure BLOB Storage for safe keeping, history, easy rollback scenarios, etc.
- Having a build number generated for a build and make sure that build number is present in the name of our packages, configs, and deployment labels in Azure for consistency
- Leveraging the Release Configurations in Visual Studio as much as possible as the designation of our different environments, since Web.Config transformations rely on this pattern already
- Being able to still package and deploy locally, in case there is an issue with the build server for some reason

Greater depth on the deployment can be found here - http://blogs.msdn.com/b/tomholl/archive/2011/12/06/automated-build-and-deployment-with-windows-azure-sdk-1-6.aspx

Here are some technical requirements to get setup with automatic deployment:

- Install Windows Azure tools on build server
- Install Windows Azure powershell cmdlets

- Download publish settings from http://windows.azure.com/downloads/publishprofile.aspx
- In MSBuild build definition setting, under process set it to
  - /t:Publish /p:PublishDir=C:\Builds\Drops\;Configuration=DevPreview;AzurePublishProfile=<Your profile>.azurePubxml

The Future of ChronoZoom

We envision a world where scientists, researchers, students, and teachers collaborate through ChronoZoom to share information via data, tours, and insight.

Imagine a world where the leading academics publish their findings to the world in a manner that can easily be accessed and compared to other data.

Imagine a tool that allows teachers to generate tours specific to their classroom needs.

This can happen with your support.   As ChronoZoom through the beta release, we need your feedback and support to continue to mold this project to suit your needs.

Help ChronoZoom evolve by taking this survey so we can provide the best possible future features: http://www.zoomerang.com/Survey/WEB22EFZBLQL4B/