Introduction

Sistema Integral Youtapit

Este sistema se compone actualmente de Ecommerce, Inventarios, y Punto de venta.

Concepto General

Personal(Usuarios)

Usuarios dentro de la empresa que administrara el sistema, se basa en diferentes areas o en una sola segun sea la estructura de la organizacion.

- [x] Creacion de usuarios
- [x] Modificacion de permisos
- [x] Eliminacion de usuarios y permisos

Activos

- [x] Listado de activos
  - [x] Asignar contraseña para acceso externo
  - [x] obtener QR para acceso externo y unico
  - [x] Datos
  - [x] Documentos
  - [x] Documentos temporales
  - [x] Enlaces
  - [x] Precarga de carta porte
  - [x] Aplicar gasto a activo.
  - [x] Reporte de gastos aplicados a activo
  - [x] Reporte de activos

Clientes

- [ ] Leads
- [ ] Prospectos
- [ ] Clientes

Administracion

- [x] Facturacion
- [ ] Cobranza
- [ ] Compras
- [ ] Pagos a Proveedores
- [x] Inventarios(activos)
- [ ] Gastos generales
- [ ] Caja chica

- [ ] Punto de Venta
- [x] Catalogos(Tipos de gastos, proveedores, etc)

Procesos

- [ ] Renta
  - [ ] Politicas de renta
- [ ] Contratos
- [ ] Entrega
- [ ] Recoleccion
- [ ] Mantenimiento
- [ ] Cotizaciones
- [ ] Ventas y Apartados

Modulos del sistema

- [ ] Clientes
  - [ ] leads
  - [ ] prospectos
  - [ ] clientes
- [x] Activos
- [ ] Cotizacion de Renta / Venta
  - [ ] Envio de correo con cotizacion
- [ ] Pedidos
- [ ] Rentas
  - [ ] Contrato
  - [ ] Cancelación
  - [ ] Extension de contrato
- [ ] Recepcion y entrega de maquinaria
- [ ] Asistencia tecnica
- [ ] Cobranza
- [ ] Requisicion de activo
- [ ] Facturacion de contratos
- [ ] CRM
- [ ] Dashboard
- [ ] Historial y seguimiento de clientes
- [ ] Chat interno
- [ ] Archivero digital
- [ ] Encuestas de calidad de servicio

Branch Strategy

Main Branches

- main (or master) Branch:
  - The production-ready code.
  - Only contains thoroughly tested and stable code.
  - Direct commits are restricted; only allowed through pull requests (PRs) after code review and approval.
- develop Branch:
  - The latest codebase reflecting the current state of development.
  - All features and fixes are integrated into this branch before being merged into main.
  - Serves as a base for all new feature branches.

Supporting Branches

- Feature Branches:
  - Naming Convention: *feature/<feature-name>*
  - Created from: *develop*
  - Purpose: For developing new features or enhancements.
  - Merging: Once complete and tested, merge back into develop.
- Bugfix Branches:
  - Naming Convention: *bugfix/<issue-id>*
  - Created from: *develop* (or *release* if the fix is for an upcoming release)
  - Purpose: For fixing bugs identified during development.
  - Merging: Merge back into develop (or release if applicable) once fixed.
- Release Branches:
  - Naming Convention: *release/<version-number>*
  - Created from: *develop*
  - Purpose: To prepare for a new production release.
  - Activities: Final testing, bug fixing, and preparing release notes.
  - Merging: Merge into both main and develop once ready.
- Hotfix Branches:
  - Naming Convention: *hotfix/<issue-id>*
  - Created from: *main*
  - Purpose: For urgent fixes that need to go directly into production.
  - Merging: Merge into both main and develop once applied.

Branch Workflow

1. Feature Development:

- Create a branch from *develop* using *feature/<feature-name>*.
- Implement the feature, commit changes, and push the branch to the repository.
- Open a pull request to merge the feature branch into *develop*.
- Conduct code reviews, perform necessary tests, and merge the changes into *develop*.

2. Bug Fixing:

- Create a branch from *develop* using *bugfix/<issue-id>*.
- Fix the bug, commit changes, and push the branch.
- Open a pull request to merge the bugfix branch into *develop*.
- After reviews and tests, merge the changes into *develop*.

3. Release Preparation:

- Create a branch from *develop* using *release/<version-number>*.
- Perform final testing, fix any last-minute bugs, and update documentation.
- Merge the release branch into both *main* and *develop* once ready.

4. Hotfixes:

- Create a branch from *main* using *hotfix/<issue-id>*.
- Apply the fix, commit changes, and push the branch.
- Open a pull request to merge the hotfix branch into *main*.
- Merge changes into *develop* to include the fix in ongoing development.

Best Practices

- **Regular Merges**: Merge develop into feature branches regularly to stay updated and avoid integration issues.
- **Code Reviews**: Conduct mandatory code reviews before merging any branch to ensure quality and adherence to standards.
- **Automated Testing**: Implement continuous integration with automated testing to catch issues early and maintain code quality.
- **Documentation**: Keep all changes well-documented, including comments in code, update logs, and comprehensive commit messages.

DEVOPS REPO AND CI / CD

Continuous Integration / Continuous Deployment

BUILD PROCESS LAYOUT

The build pipelines are responsible for building and running automated unit / integration testing of the product changes. They are set to trigger automatically and run locally on MBCI servers. The results of the build are stored in a staging location, waiting for the release process to pick them up. Staging of product changes are on the MBCI build server under the STAGE folder.

RELEASE PROCESS LAYOUT

The release pipelines are responsible for testing and deploying the product to the associated servers and databases. This will pick up the STAGE build files and move those changes to the respective environment. The release is triggered by a sprint build which then deploys those changes to the testing environment. Chained to the testing environment deployment is the release to production. After the UAT cycle has been approved, then the PROD deployment starts. These are deferred deployments for the same evening or early next morning.

WORKFLOW FOR DEVELOPER

All coding change work should be conducted in the correct feature branch using their name and SNOW number. This follows the standard naming convention for feature branches. Work is checked into source control under the feature branch at the end of the shift to ensure all code is available and stored safely.

The following outlines the typical development cycle

1. Stories are assigned to a sprint.
2. Changes are made to code in source control for the respective product repo.
3. In-complete code changes are checked into source control under the story or feature branch.
    1. Pick up the PBI where work is to be completed. (Work only on 1 PBI at a time)
        1. Ensure the state of the PBI is Development (if not already completed).
        2. Change the state of your task from To Do to In Progress
    2. If switching to a new PBI
        1. Make sure you follow the story or feature branch naming convention to keep work separated from one PBI to the next.
4. Make changes to the product per the PBI and Task item descriptions in the feature branch.
5. Submit a pull request for code review when coding changes are complete
6. Change the state of the Task item to Done
7. Move the PBI to the next state when all task items have been completed. This will typically be business testing.

PIPELINES FOR LIBRARIES

When making changes to a library product that ends up being part of a NuGet package. Follow these versioning rules so that the product versions can be referenced and organized.

Our versioning follows this pattern Major.Minor.Patch.

In the build pipeline yaml file a version number is maintained and the following rules should be applied to update this file.

- Major = is incremented when a breaking change is made, or a significant change set is applied to the library.
- Minor = is incremented when a new feature is implemented for the library based upon the story making the change. Minor is reset to zero when the major is incremented.
- Patch = is automatically incremented for each build by the system and should not need to be updated.

AGILE PROCESS

SCRUM TEAMS

The manufacturing application team is the scrum team doing the work. There are multiple teams under the MFGApplications team project (teams of teams). The team resources work from the MFGApplications team boards, which encompasses all the sub teams. The sub teams are organized by product (FIMA, YouTapIt,Other) and exist for operations awareness.

INCIDENT RESOLUTION

Production related incidents are managed by the team for the product they support. We do not have separate resources to manage release hot fixes. Incidents are added to the backlog as bug items and tracked as requirements. As incidents are raised in service-now they will follow 1 of two paths: 1) P1 and P2 tickets require immediate resolution on current sprint, 2) P3 and below are planned for future sprints. Unplanned work may affect the team commitment for sprint items, so adjustments will need to be made based upon capacity.

ADDING INCIDENTS TO THE BACKLOG

1. SNOW incidents are entered in the backlog as a BUG
    1. The SNOW ticket should be added to the SNOW field
    2. The caller from the SNOW ticket should be added to the Business Owner field
    3. Relative effort sizing based upon the developer opinion will need to be added to the effort field
    4. The description should be copied from the SNOW ticket
2. All incident BUGs should be added to the TOP of the backlog
3. If the incident is not added to the current sprint the SNOW ticket state should be set to planning

DAILY STANDUP SCRUM MEETING

The scrum meeting is usually scheduled for 15 mins.

The following activities are discussed during the daily scrum meeting:

- What I completed yesterday to achieve sprint goals
- What I am working on today to achieve sprint goals
- What is impeding me from completing my work.

WORK ITEM TYPES

PRODUCT BACKLOG ITEM

This is the default work item type in our agile boards. It is used to keep track of the business change request and typically applies to enhancement work.

INCIDENTS

Incidents are production related issues and identified as a BUG work item type. They come from service now and tracked in the team board.

CHANGE REQUEST

This is a specialized work item type used to keep track of changes being deployed. It is mapped to a specific change request that has related work items attached to it. Those items are specified on the change ticket in SNOW. It will show the release names on the release tab in TFS.

BUGS

Bug items identify an issue with work that was completed from features. These typically are entered by the business and identify the issue along with steps to reproduce. They are used to keep track of the fixes required for the related work item. They can come into a sprint while the business is testing.

SWIM LANES

We currently do not use swim lanes.

SPRINT DEVELOPMENT

Stories is regularly groomed during throughout each cycle. Priority items are groomed first and staged onto future sprints. When a new sprint is being created, these work items are assigned to a sprint and team members based upon available capacity.

Each of the work items is organized in priority order (top to bottom) and should be worked on in that order. This will keep the team aligned with business need as well as aligning interdependent work across

the team.

As a team member, when your task item becomes available for work to be completed, review the description, and complete the work outlined.

All work should be completed by the end of the sprint. This will keep the entire team on track with the timelines communicated to the business.

If new work is required mid sprint, the team can negotiate what adjustments may need to be made to current commitments.

SOURCE CONTROL

DATABASE PROJECTS

All MFG stack applications will have a database projects in source control. These projects are used to update our databases with the latest changes in an automated fashion. The code in the database project is KING when it comes to what is in the physical database. If changes are applied manually to the database, these can/will get wiped out the next time the database is published. It is imperative that ANY change needed for the database is coded in the database project otherwise those changes will be lost forever.

All projects have a schema compare object that will take updates from the database and put them in the database project. You can use these to keep source control up to date with the database (if needed). They should be used only when needed as all development work should be done in the project itself.

In some database projects there is a reference projects which are shared between one database and another. These are classified as a composite database (the database is split between 2 different projects within the solution). These projects should only have objects that are shared between the databases.

DEVELOPER PROCEDURES

WORKFLOW

The following outlines our processes to manage the work requirements. This outline does not cover every conceivable situation but does outline a typical cycle from product inception to production.

NEW WORK

1. PO / BA (In DevOps)
    1. Creates PBI
    2. Prioritizes PBI to business delivery schedule
    3. Moves PBI to grooming state when ready to take work on
        1. Grooms PBI with requirements
    4. Grooms backlog requirements with team

    1. Team provides effort estimates
    2. Moves PBI to planning column
2. PBI's in the planning column are staged for sprint assignment
    1. Tasks are created with hour estimates added for PBI's by the team.

SPRINT DEVELOPMENT CYCLE

1. When sprint starts, PBI's are moved to the development state.
2. Development Team (in SNOW) a. Changes state of ticket to: i. State = Work In Progress ii. Sub State = Development in Progress
3. Development Team (In DevOps) a. Picks up task items and develops work to complete PBI b. Send completed work for code review through Pull Request i. Reviewers look at code and provides developer feedback or approves changes. Stack lead must be one of the reviewers. ii. Developer makes any necessary changes and resubmits pull request until everything passes c. Work is checked into source control for respective feature branch i. Each feature branch should be linked to a specific PBI and contain changes for that PBI. NEVER develop features for other PBI's onto other feature branches. ii. Update the comments for the check to state what the reason for the check in is. d. Update the task item for the work is DONE. e. When ALL development task items are complete for a PBI and ALL code reviews are clean move the PBI to the User Testing column.
4. Development Team (In SNOW) a. Change ticket sub state to QUA required when all required changes for a PBI are complete RELEASE DEVELOPMENT WORK
5. Business (In SNOW) a. After business tests and approves the work attach test results to the SNOW ticket b. Change the ticket sub state to QUA Approved
6. Business (In DevOps) a. Move the PBI to the Deployment Column
7. Development Team (In SNOW) a. Create SNOW change ticket i. Associate ER tickets to change ticket b. Updates change ticket with deployment plan i. CI/CD Automated ii. Submit the change ticket for approval
8. Development Team (In DevOps repo) a. Only changes that have business approval and change control approval in SNOW can continue in this process. b. Only work items in the "Deployment" column can continue in this process. c. Create a release branch for the affected repos. d. Verify ALL required changes have moved correctly into the main trunk. i. Open associated projects and build on local machine. Fix any changes that may need to be made.
9. Development Team (In DevOps) a. Create Change Request work item for release i. Relate PBI's to Change Request b. Update Change Request with release details c. Update Change Request with SNOW change ticket number d. Move Change Request to Deployment column PRODUCTION RELEASE
10. Development Team (In SNOW) a. Updates change ticket as complete b. Moves PBI's to Done column BUG REMEDIATION When the business is testing product changes, some issue may occur. These should be reflected in a BUG work item and associated to the PBI being tested. The BUG should include steps on how the issue was encountered and list the expected results.
11. Business (In SNOW) a. Reject ER ticket as re-work

12. Business (In DevOps) a. Creates a bug item for the PBI being tested b. Updates the steps and acceptance criteria
13. Stack Lead (In DevOps) a. Evaluates bug item validity and defines actions b. Bug work item assigned to sprint for re-work c. Changes state to Approved
14. Developer (In DevOps repo) a. Creates a BugFix branch for the bug. b. Reworks PBI changes to account for BUG i. Associate rework to PBI and BUG work item c. Creates a pull request for code review d. Change state of BUG to Committed
15. Review of pull request is completed a. Developer changes state of SNOW ticket to QUA Required b. Bug fix branch is merged back into main branch.
16. Stack Lead (In DevOps) a. After code review complete changes BUG state to Done CODE REVIEWS When requesting a code review, create a pull request and associate it to the specific PBI and Task item the changes relate to. This will help streamline the review process so that the reviewer knows what to specifically look at. Also, the senior developer must be a required reviewer on the pull request. The pull must include one other developer, but they may not be required. MIGRATION TO PROD Production migration occurs when a story or feature is approved through the automated release pipeline. These changes are carried from the sprint branch to production. As product is approved by the business and the change request is approved by leadership, the release pipeline can be updated and approved for release to production that evening or early the next morning. It is the developer responsibility to ensure that product changes moving into the release branch is complete and accurate. CODING PRACTICES The manufacturing application team uses sandcastle to document our product. This will take XML comments from the compiled library and build a documentation website. To ensure our documentation is complete and accurate the following rules apply: • All classes, properties, members, and methods should have xml comments • Comments should include standard and enhanced (SHFB) nodes (see references) • Use a NamespaceDoc class to document namespaces • Use a NamespaceGroupDoc class to document namespace groups • Revision history should be included at the class and method levels o The details should include a description of what was changed. o A detailed list can be provided which can help provide clarity of the changes made • Class and Method implementation should be provided to help other developers understand how to use the class or method. CODING STANDARDS The standard outlined below is intended to be a guideline for all internal and contracted resources. Adherence to the designated standards of MasterBrand Cabinets, Inc. will make knowledge transfer/distribution among the internal programming staff and external contracted vendors easier and more uniform.
    The following standard has code samples presented in C#. Note that this language preference is not required, but the methodology outlined with code samples is. If shifting to .NET from another coding language, then the preferred language to learn and develop is C# if there is no previous background in .NET programming. This initial version is also intended to provoke discussion within MasterBrand Cabinets, Inc and in the valued contractors that are utilized. Any questions, additions, or modifications will be discussed and consensus met prior to modifying this document. Once changes are made, the version number will reflect any change to the standards document. *Note

that all code samples are evident below through the use of italics. *Note that examples are evident below in Times New Roman font. NAMING CONVENTIONS AND STYLES

17. Use Pascal case for type (class, structure, etc), method names, and constants.

```
Public class SomeClass
{
        const int DefaultSize = 10;
        public void SomeMethod()
        {}
}
```

3. Use camel case for local variable names and method arguments.

```
Int number;
Void MyMethod(int, someNumber)
{…}
```

5. Prefix interface names with I

```
Interface IMyInterface
{…}
```

7. Prefix private member variables with _. Use Pascal case for the rest of the variable name following the _.
8. Suffix custom attribute classes with Attribute.
9. Suffix custom exception classes with Exception.
10. Name methods using verb-object pair. (i.e. GetProductPrice(), ValidateProductModification(), GetCustomerMultiplier())
11. Use descriptive variable names. a. Avoid single variable names, such as i or t. Use index or temp. b. Avoid using Hungarian notation for public or protected members. c. Do not abbreviate words such as num for number.
12. Always use C# predefined types rather than their aliases in the System namespace. Example:

```
object NOT Object
string NOT String
int NOT Int32
short NOT Int16
```

14. Use descriptive namespaces such as company name or the product name. Example: If the application name is eQuote and the Company name is MBCI. The namespace would be:

```
MBCI.eQuote.DataAccessLayer
MBCI.eQuote.Common
```

16. Avoid fully qualified type names. Use the using statement.

```
// Correct:
using System.Configuration;
ConfigurationManager.GetSetting("dbConn");
// Incorrect:
System.Configuration.ConfigurationManager.GetSetting("dbConn");
```

18. Avoid putting the using statement inside the namespace.
19. Put custom or third-party namespaces underneath framework namespaces.
20. Use delegate inference instead of explicit delegate instantiation.

```
Delegate void SomeDelegate();
public void SomeMethod()
{…}
SomeDelegate someDelegate = SomeMethod;
```

22. Maintain indentation. No tabs or single spaces. Uniform indentation should span the entire project.
23. Comment indentation should be in line with the code being written.
24. Misspelled comments allude to messy code. Spell check all comments.
25. All member variables should be declared at the top with one line separating the variable(s) from all properties and methods.

```
Public class SomeClass
{
        private int _CustomerId;
        private string _CustomerName;

        public void SomeMethod1()
        {…}
        public void SomeMethod2()
        {…}
}
```

27. Declare a local variable as close to first use as possible.
28. A file name should reflect the class within.
29. Always place an open curly brace ({) on a new line.
30. Declare each variable independently on separate lines.

31. Group class implementation in the following order: a. Member variables b. Constructors & Finalizers c. Properties d. Methods i. Instance Methods ii. Static Methods
32. Prefix Boolean variables with “Can”, “Is”, or “Has”.

```
bool IsCustomerActive;
bool CanCustomerOrder;
bool HasCreditAvailable;
```

CODING GUIDELINES

1. Class files cannot consist of multiple classes. One class per class file.
2. Attempt to limit files to 500 lines or less.
3. Methods should consist of 50 lines or less.
4. Avoid methods with more than 10 arguments. Use structures for passing multiple arguments.
5. Do not manually edit any machine-generated code. a. Proxy classes generated using “wsdl” or “Add Web Reference”. Do not modify the constructor of these classes to communicate with a different URL or add credential information. Set all these properties from the code that instantiates the proxy class.
6. Each page should contain header comments detailing the original developer, a data time stamp, and brief summary of the page functionality.
7. Additions/changes to existing pages should contain additional comments at the header comment level underneath the original with the developer name, date time stamp of change, and description of that change.
8. Avoid obvious comments. Variable and method names should be sufficiently descriptive to all developers.
9. Never hard code a numeric value. Always declare a constant.
10. Never use const on read-only variables. Use the readonly directive.

```
public class MyClass
{
        public const int DaysInWeek = 7;
        public readonly int Number;
        public MyClass(int, someValue)
        {
                Number = someValue;
        }
}
```

11. Assert every assumption. Typically every fifth line is an assertion.

```
using System.Diagnostics;
object GetObject()
{…}
object someObject = GetObject();
Debug.Assert(someObject != null);
```

13. Catch exceptions for which there is explicit handling. If an exception is caught, always throw the original exception or another exception constructed from the original exception, to maintain the stack location of the original error.

```
catch(Exception exception)
{
        MessageBox.Show(exception.Message);
        throw;
}
```

14. Do not use error codes as function return values.
15. Limit custom exception classes. Custom exception classes should follow these rules: a. Derive the custom exception from Exception b. Provide custom serialization
16. Limit public types to what is necessary. All others should be marked internal.
17. Use class libraries to contain business logic.
18. Do not use explicit values for enums.

```
// Correct
public enum Cabinet
{
        Base, Wall, Tall
}
//Incorrect
public enum Cabinet
{
        Base = 0, Wall = 1, Tall = 2
}
```

19. Do not specify a type for an enum.

```
// Avoid
public enum Cabinet:long
{
        Base, Wall, Tall
}
```

20. If statements require the use of a curly brace.
21. Do not use the ternary conditional operator. Bool IsValid = PerformValidation() ? true:false;
22. Do not use Boolean calls in conditional statements. Use local variables.

```
bool IsValidationSuccessful()
{…}
// Avoid
If(IsValidationSuccessful())
{…}
// Correct
bool ok = IsValidationSuccessful();
if(ok)
{…}
```

23. Use zero-based arrays.
24. Explicitly initialize an array of reference types using a for loop.

```
Public class MyClass
{
        const int ArraySize = 100;
        MyClass[] array = new MyClass[ArraySize];
        For(int index = 0; index < array.Length; index++)
        {
                Array[index] = new MyClass();
        }
}
```

25. Do not provide public or protected member variables. Use properties.
26. Do not use the inheritance qualifier. Use override.
27. Mark public and protected methods as Virtual in a non-sealed class.
28. Do not use unsafe code, except when using interop.
29. Avoid explicit casting. Use the “as” operator to defensively cast to a type.

```
Model model = new Cabinet();
Cabinet cabinet =model as Cabinet;
If(cabinet != null)
{…}
```

30. Check a delegate for null before invoking.
31. Do not hardcode strings that are presented to end users. Use resources or configuration files.
32. Do not hardcode a string that could change based on deployment.
33. Use String.Empty, not “”.

34. Use String.Compare, not the “==” operator.
35. Use StringBuilder when building long strings.
36. Do not concatenate strings inside a loop.
37. Use String.Length = 0 property to check for empty strings.
38. Implement Object.ToString() to return a textual expression.

```
public override string ToString()
{
        return String.Format(“CustomerId: {0}”, CustomerId);
}
```

39. Use application logging and tracing.
40. Always have a default case in a switch statement.

```
int number = SomeMethod();
switch(number)
{
        case 1:
                Trace.WriteLine(“Case 1:”);
                break;
        case 2:
                Trace.WriteLine(“Case 2:”);
                break;
        default:
                Debug.Assert(false);
                break;
}
```

FRAMEWORK GUIDELINES

1. Use type-safe data sets or data tables. Avoid raw ADO.NET.
2. Always use transactions when accessing a database. a. Use Enterprise Services or System.Transactions transactions. b. Do not use ADO.NET transactions by enlisting the database explicitly.
3. Use transaction isolation level set to Serializable.
4. Do not use the Data Source window to drop connections on Windows Forms, ASP.NET forms, or Web Services. Maintain separation between the presentation tier and the data tier.
5. Do not use SQL Server authentication. If it is unavoidable to use SQL Server authentication, the database connection string must be encrypted.
6. Do not put logic within a stored procedure. Outside of simple switching logic to vary a query based on parameter values, logic should be placed in the business logic layer of the code.
7. No code should be placed in the ASPX files. All code should be placed in code-behind pages.

8. Check all session variables for null prior to accessing.
9. Avoid setting the AutoPostBack property of server controls to true.
10. Turn on SmartNavigation for ASP.NET pages. This will reduce “flicker” on post back.
11. Always provide namespace and service descriptions for web services.
12. Always provide descriptions for web methods.
13. When adding a web service reference, provide a meaningful name for the location.
14. In ASP.NET pages and web services, the session variables should be wrapped in a local property. Only that property should be allowed to access the session variable, and the rest of the code should use the property, not the session variable.

```
public class Calculator:WebService
{
        int Memory
        {
                get
                {
                        int memory = 0;
                        object state = Session[“Memory”];
                        if(state != null)
                        {
                                Memory = (int) state;
                        }
                        return memory;
                }
                set
                {
                        Session[“Memory”] = value;
                }
        }
        [WebMethod(EnableSession = true)]
        public void MemoryReset()
        {
                Memory = 0;
        }
}
```

15. Always dispose of a TransactionScope object.
16. Do not put any code after the call to Complete() inside a transaction scope.
17. Do not set the transaction timeout to zero in Release builds. Set infinite timeout.
18. Do not catch exceptions in a transactional method. Use the AutoComplete attribute.
19. Do not call SetComplete(), SetAbort(), and any similar. Use the AutoComplete attribute.
20. Always override CanBePooled and return true.
21. Always call Dispose() explicitly on pooled objects unless the component is configured to use JIT.

22. Never call Dispose() when the component uses JIT.
23. Always set authorization level to the application and component.
24. Set authentication level to privacy on all applications.
25. Set impersonation level on client assemblies to Identity.

REFERENCES • XML documentation comments: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/ • Sandcastle XML comments Guide: https://ewsoftware.github.io/XMLCommentsGuide/html/4268757F-CE8D-4E6D-8502-4F7F2E22DDA3.htm •