Sage 300 Web Screens SDK

Coding Standards

December 2016

This is a publication of Sage Software, Inc.

Copyright © 2016. Sage Software, Inc. All rights reserved.

Sage, the Sage logos, and the Sage product and service names mentioned herein are registered trademarks or trademarks of Sage Software, Inc. or its affiliated entities. All other trademarks are the property of their respective owners.

Business Objects® and the Business Objects logo, BusinessObjects®, and Crystal Reports® are trademarks or registered trademarks of Business Objects Software Ltd. in the United States and in other countries. Business Objects is an SAP company.

Microsoft ®SQL Server®, Windows Vista® and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or in other countries. The names of all other products and services are property of their respective owners.

Your use of this document and the Sage product(s) described herein is governed by the terms and conditions of the Sage End User License Agreement ("EULA") or other agreement that is provided with or included in the Sage product. Nothing in this document supplements, modifies or amends those terms and conditions. Except as expressly stated in those terms and conditions, the information in this document is provided by Sage "AS IS" and Sage disclaims all express, implied or statutory warranties of any kind, including but not limited to the warranties of merchantability, fitness for a particular purpose or of noninfringement.

No implied license is granted to you under any Sage intellectual property or trade secret right. Sage reserves the right to revise, supplement or remove information in this document at any time without notice to you or others.

Sage End User License: [www.sage.com/us/legal/eula](http://www.sage.com/us/legal/eula)

Contents

[1. Overview 5](#_Toc440459703)

[1.1 All-In-One Code Framework Coding Standards 5](#_Toc440459704)

[1.2 Software Coding Standards Document 5](#_Toc440459705)

[2. Data Types 6](#_Toc440459706)

[2.1 Alias Data Types 6](#_Toc440459707)

[2.2 Implicitly Typed Local Variables 6](#_Toc440459708)

[3. Namespaces 8](#_Toc440459709)

[3.1 Namespace Segments 8](#_Toc440459710)

[4. Comments 9](#_Toc440459711)

[4.1 XML Comments 9](#_Toc440459712)

[4.2 Copyright Comments 10](#_Toc440459713)

[5. Naming Conventions 11](#_Toc440459714)

[5.1 Redundancy 11](#_Toc440459715)

[5.2 Descriptive Names 12](#_Toc440459716)

[5.3 Method Parameter Names 13](#_Toc440459717)

[5.4 Parameter Driven Generic Routines 13](#_Toc440459718)

[5.5 Technical Terms 14](#_Toc440459719)

[5.6 Company and Product Name 15](#_Toc440459720)

[6. Razor Views 16](#_Toc440459721)

[6.1 Widget Attributes 16](#_Toc440459722)

[6.2 Buttons with Images 16](#_Toc440459723)

[6.3 ID Attribute 17](#_Toc440459724)

[7. Abbreviations 18](#_Toc440459725)

[7.1 Approved Abbreviations and Acronyms 18](#_Toc440459726)

[8. Javascript 23](#_Toc440459727)

[8.1 Using Strict Mode 23](#_Toc440459728)

[8.2 Cache your objects 23](#_Toc440459729)

[8.3 Define and using Enum 24](#_Toc440459730)

[8.4 Don’t Repeat Yourself (DRY) 25](#_Toc440459731)

[8.5 Using “===” and “!==” for primitive values compare 27](#_Toc440459732)

[8.6 Magic Numbers 27](#_Toc440459733)

[8.7 Comment code appropriate 27](#_Toc440459734)

[9. Jquery 28](#_Toc440459735)

[9.1 Selectors 28](#_Toc440459736)

[9.2 DOM Insertion 28](#_Toc440459737)

[9.3 Caching 29](#_Toc440459738)

[9.4 Chaining 29](#_Toc440459739)

[9.5 Object Literals 30](#_Toc440459740)

[9.6 CSS 30](#_Toc440459741)

[9.7 Avoid multiple $(document).ready() calls 31](#_Toc440459742)

[9.8 this vs. $(this) 31](#_Toc440459743)

[9.9 Use Sub-queries 31](#_Toc440459744)

[9.10 Leverage Event Delegation (a.k.a. Bubbling) 32](#_Toc440459745)

[9.11 AJAX Methods 33](#_Toc440459746)

[9.12 Passing an Attribute Object 33](#_Toc440459747)

[9.13 Use live() 34](#_Toc440459748)

[9.14 Use jQuery.noConflict() 34](#_Toc440459749)

1. Overview
   1. All-In-One Code Framework Coding Standards

The *All-In-One Code Framework Coding Standards* document describes the coding style guideline for native C++ and .NET (C# and VB.NET) programming used by the Microsoft All-In-One Code Framework project team.

The Sage 300 Web UI project is implemented in the ASP.NET MVC 4.0 framework using C#. Therefore, we are only concerned with the .NET C# standards.

* 1. Software Coding Standards Document

This document is meant to supplement standards found in the *All-In-One Code Framework Coding Standards* document. As a rule, the guidelines in the base document are to be followed unless a guideline in this document overrides or enhances a standard found in the base document.

The standards found in this document are not necessarily Sage 300 specific, but have been adopted as standards for use by the Sage 300 development team (internal, external or third party) for the purposes of:

* Understandability

The code must be clearly readable and straightforward. It must be documented with XML comments and comments for any code requiring additional clarification.

* Correctness

The code must behave as documented.

* Consistency

The code must follow all established coding standards. It should not be discernible as to what type of developer created the code (internal, external, etc.) or which developer created the code.

* Modernization

The code must adopt current standards and practices found in the language

1. Data Types

A consistent use of data types provides a consistent look and feel to the software. Therefore, alias data types and implicitly typed local variables are required.

* 1. Alias Data Types

The .NET language provides alias data types for the system data types (i.e. string for System.String, int for System.Int32, etc.).

It is best to use the alias data type as opposed to the fully qualified data type, or even “String” in the example of the string data type. Use of the alias data type is a way to avoid type conflicts with any other included libraries.

Also, alias data types are reserved words, whereas system data types are just class names. This means that the alias data types cannot be used as variable names whereas the system data types can (not that you would, but you could).

**Bad:**

public String GetSomeString()

{

return "Some string";

}

**Good:**

public string GetSomeString()

{

return "Some string";

}

* 1. Implicitly Typed Local Variables

Local variables can be given an inferred “type” of var instead of an explicit type. The var keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement.

For more information on the use of the var implicitly typed local variable, see <http://msdn.microsoft.com/en-us/library/vstudio/bb384061.aspx>.

Additionally, CNA developers will be using the ReSharper productivity tool, which finds and highlights errors and code smells. The use of the var syntax is one of the code suggestions presented to the user when an explicitly typed local variable is used. Also, ReSharper will not suggest an implicitly type local variable when one is not needed or cannot be implemented.

**Bad:**

public string GetSomeString()

{

string retVal = Resources.Common.AclCustomer;

…

return retVal;

}

**Good:**

public string GetSomeString()

{

var retVal = Resources.Common.AclCustomer;

…

return retVal;

}

1. Namespaces

Namespaces are used in .NET code to organize classes and to control the scope of a class and method names in larger projects.

For more information on namespaces, see <http://msdn.microsoft.com/en-us/library/vstudio/0d941h9d.aspx>.

* 1. Namespace Segments

The namespace will contain seven segments:

{Company}.{Region}.{Division}.{Type}.{Product}.{Application}.{Name}

* Company

Sage

* Region

CA (for Canada), US (for United States)

* Division

SBS (Sage Business Solution)

* Type

ERP (Enterprise Resource Planning)

* Product

Sage300 (Sage 300 Product)

* Application/Module

AS (for Administrative Services), AP (for Accounts Payable), etc.

* Name

{Descriptive Name} (i.e., Resources, Services, Models, etc.)

**Bad:**

namespace Sage.Canada.SBS.ERP.Sage300.AS.BusinessRepository

namespace Sage.CA.SBS.ERP.Sage300.Services

Good:

namespace Sage.CA.SBS.ERP.Sage300.AS.BusinessRepository

namespace Sage.CA.SBS.ERP.Sage300.AP.Services

1. Comments

Refer to the *All-In-One Code Framework Coding Standards* document for good use of comments.

The information in the following section overrides, enhances, and/or adds to the standards presented in that document.

* 1. XML Comments

XML comments are to be used for class definitions, public and private class variables and properties, and public and private methods.

The *All-In-One Code Framework Coding Standards* document states that “all public and non-public functions that are not trivial” should be commented. **However, the standard for Sage 300 is that all routines be commented regardless of their triviality.**

The intention is to not only provide a better experience for the consumer of our public resources, but for the internal developer as well, who will have to code and reference more than public routines and properties.

Bad:

public Vendor Next(string currentId)

{

return Navigate(currentId,!string.IsNullOrEmpty(currentId));

}

Good:

/// <summary> Get Next from repository </summary>

/// <param name="curntId">Current ID</param>

/// <returns>Next Vendor</returns>

public Vendor Next(string currentId)

{

return Navigate(currentId,!string.IsNullOrEmpty(currentId));

}

The XML comments for public and private routines must specify the <summary>, <param> (if any), and <return> tags (if the routine returns a value). Also, the <remarks> tag is a good place to store additional information about the routine, which may not be appropriate to include in the <summary> tag.

The XML comments for generic classes must specify <typeparam name="T">.

/// <summary>

/// View Model for Accounts View

/// </summary>

/// <typeparam name="T">Account Type View Model</typeparam>

public class AccountTypeViewModel<T> : ViewModelBase<T> where T : Accounts, new()

{

//Code here..

}

Sage 300 developers use the ReSharper productivity tool, which finds and highlights errors and code smells. Failure to include correct parameter comments is noted by the tool.

* 1. Copyright Comments

All C# (.cs), JavaScript (.js) and razor View (.cshtml) code files must have copyright comments.

.cs and .js files:

// Copyright (c) 1994-2016 Sage Software, Inc. All rights reserved.

.cshtml file:

// Copyright (c) 1994-2016 Sage Software, Inc. All rights reserved.

1. Naming Conventions

Naming conventions or standards allows for the code to be structured, easily read and consistent. Consistency between development groups (internal vs. external) and consistency between developers is the goal of this section. It should not be discernible as to which developer created the code.

Refer to the *All-In-One Code Framework Coding Standards* document for good use of naming conventions. The following section overrides, enhances, and/or adds to the standards presented in that document.

* 1. Redundancy

Within a namespace, the names of methods, properties, variables, etc. should not make redundant use of the application prefix and/or repository name. This only adds clutter and there is no reason to explicitly add this additional information.

Additionally, there will be “template” type of code which will be found in every controller, every model, every view, etc. Once a developer has seen a template method, property, variable in one class, the goal is to have a consistent experience in the other classes. Explicit names make the consistency more difficult to see, harder to “copy and paste”, more difficult if an automated tool needs to modify code, and so on.

However, there may be exceptions if a repository has multiple entities (such as GetBatches(), GetInvoices(), and so on).

Bad:

#region Business View Variables

/// <summary> AP0015 - Vendor header details </summary>

private BusinessEntity \_vendorHeader;

/// <summary> AP0407 - Vendor Optional fields </summary>

private BusinessEntity \_vendorDetail;

/// <summary> AP019 - Vendor Statistics </summary>

private BusinessEntity \_vendorStatistics;

/// <summary> AP014- Vendor Comments </summary>

private BusinessEntity \_vendorComments;

#endregion

Good:

#region Private Variables

/// <summary> AP0015 - Header </summary>

private BusinessEntity \_header;

/// <summary> AP0407 - Optional </summary>

private BusinessEntity \_optional;

/// <summary> AP019 - Statistics </summary>

private BusinessEntity \_statistics;

/// <summary> AP014- Comments </summary>

private BusinessEntity \_comments;

#endregion

Bad:

/// <summary> Save Vendor data </summary>

/// <param name="vendor">Vendor</param>

/// <returns></returns>

public Vendor SaveVendor(Vendor vendor)

{

Good:

/// <summary> Save Vendor</summary>

/// <param name="vendor"> Vendor </param>

/// <returns>Vendor</returns>

public Vendor Save(Vendor vendor)

{

* 1. Descriptive Names

Method, property and variable names should be descriptive, generic and agnostic without adding redundant information (see the previous section) and parameter descriptions for methods.

The name should not reflect the parameters, if any, and overloaded method names should be used instead of descriptive unique method names.

Bad:

retVal = SearchOnVendorNumber(Convert.ToString(id, CultureInfo.InvariantCulture));

Good:

retVal = Search(vendorNo);

vendor = Search(vendorNo);

Bad:

previousVendor = GoPreviousVendor(currentVendorNumber);

Good:

vendor = Navigate(currentVendorNumber, false);

Bad:

BrowseOnVendorNumber(vendorNumber, ComposeViewGroup.VendorData);

Good:

Search(\_header, Vendor.Fields.VendorId, vendorNumber, true, 0,

ViewFilterOrigin.FromStart, false);

Bad:

public Vendor GetProcessingDefaultsByGroupCode(Vendor vendor)

{

Good:

public Vendor GetDefaults(Vendor vendor)

{

* 1. Method Parameter Names

Method parameter names that are IDs should have generic name instead of detailed name.

The parameter name should be simple and consistent across the application.

Bad:

public SegmentCodeViewModel<T> Get(string accountNumber)

{

Good:

public SegmentCodeViewModel<T> Get(string id)

{

* 1. Parameter Driven Generic Routines

In the case where routines contain identical logic in numerous entities or repositories, with the exception of the entity or repository itself, a parameter driven generic routine should be created and implemented wherever possible (mostly in base classes (help section) or in help classes) instead of creating specific routines in every entity or repository.

Bad:

retVal = SearchOnVendorNumber(Convert.ToString(id,

CultureInfo.InvariantCulture));

Good:

retVal = Search(vendorNumber);

vendor = Search(vendorNumber);

Bad:

BrowseOnVendorNumber(vendorNumber, ComposeViewGroup.VendorData);

Good:

Search(\_header, Vendor.Fields.VendorId, vendorNumber, true, 0,

ViewFilterOrigin.FromStart, false);

* 1. Technical Terms

Method, property, and variable names should not include technical terms. Use verbs or verb phrases to name methods.

Bad:

retVal.Errors = GetErrorList();

Good:

retVal.Errors = GetExceptions();

Bad:

Processing = LoadVendorProcessing(view),

Good:

Processing = GetVendorProcessing(view),

Below are other technical names that can be avoided.

1. Paged
2. Data
3. Line
4. Detail
5. Details
6. Retrieve
7. Fetch
   1. Company and Product Name

The Company Name and Product Name should not be included in the names of methods, properties, or variables. The usage of these terms complicates rebranding efforts and introduces unnecessary company and product specific names.

Bad:

public SageEnumerableResponse<Vendor> Get(string filter = null, OrderBy

orderBy = null)

{

Good:

public EnumerableResponse<Vendor> Get(string filter = null, OrderBy

orderBy = null)

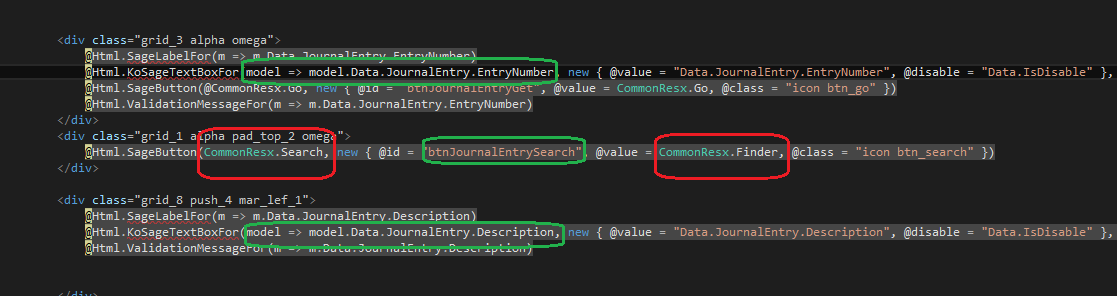
1. Razor Views

Razor Views are the “V” in Microsoft’s MVC pattern. This section lists standards associated with Razor Views.

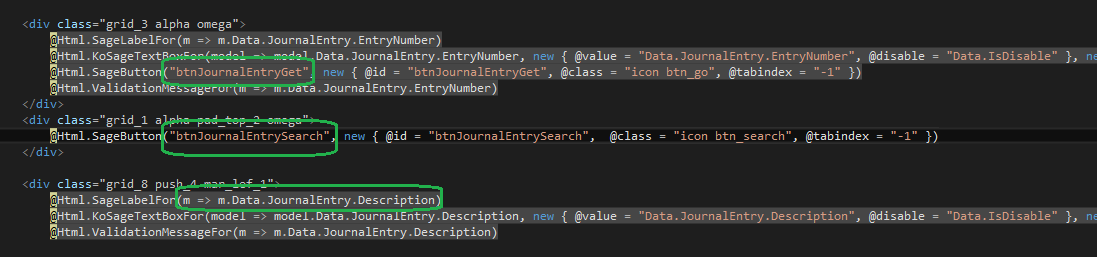
* 1. Widget Attributes

Widget attributes are assigned as either parameters or HTML attributes of the parameter list. The name of the widget is to either come from a literal or from the property name of the model. Do not use resource strings, as these strings are localized.

Bad:



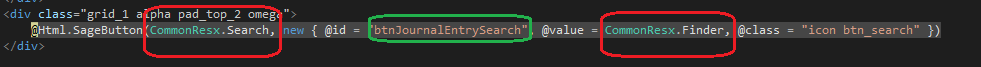
Good:

****

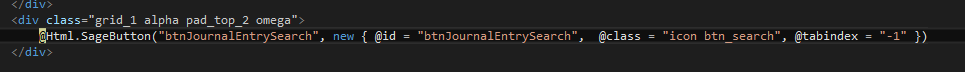
* 1. Buttons with Images

Buttons that have images are controlled from CSS and do not need to have the resource string specified, as the value property of the button will not be displaying text. Therefore, do not add the resource string to the attribute list.

Bad:

****

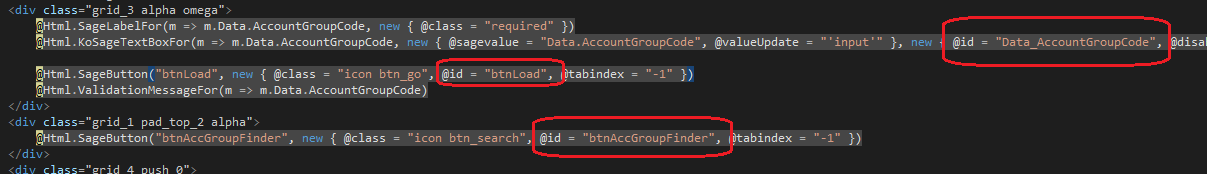
Good:

****

* 1. ID Attribute

The id attribute is to ALWAYS be present when creating widgets in the Razor View.

Good:

****

1. Abbreviations

Abbreviations used in method names, class names, property names, variables etc. can allow for these usages to be meaningful when the name is too long or an acronym or abbreviation is better suited.

However, as a general rule, abbreviations, and acronyms that are not widely accepted, should be avoided and used only when necessary. For more information, see <http://msdn.microsoft.com/en-us/library/vstudio/ms229045(v=vs.110).aspx>.

Necessity is defined as when the name exceeds 25 characters, or when the use of an abbreviation or acronym is appropriate.

Remember: clarity trumps brevity. The names are meant to be clear to anyone looking at the name; they should not have to guess at the meaning. The following list provides approved abbreviations and acronyms to be used where and when it is appropriate to do so.

* 1. Approved Abbreviations and Acronyms

| Abbreviation | Meaning | Abbreviation | Meaning |
| --- | --- | --- | --- |
| Acct | Account | Accts | Accounts |
| Ack | Acknowledgment | Act | Actual |
| Addl | Additional | Addr | Address |
| Adj | Adjustment | Amt | Amount |
| App | Application | Appr | Approval |
| Auth | Authorization | Asst | Assisted |
| Avail | Available | Auto | Automatic |
| Bal | Balance | Avg | Average |
| Beg | Beginning | Bdgt | Budget |
| BOL | Bill of Lading | BOM | Bills Of Material |
| Bus | Business | Cal | Calendar |
| Calc | Calculation | Cat | Category |
| Cert | Certification | Chk | Check |
| Chks | Checks | Chrg | Charge |
| Clr | Clearing | Cmnt | Comment |
| Cnt | Count | Cntr | Counter |
| Cntrb | Contribution | Col | Column |
| Comb | Combine | Comm | Commission |
| Comp | Component | Const | Constant |
| Conv | Conversion | COS | Cost of Sales |
| Cr | Credit | CrCard | Credit Card |
| Ctrl | Control | Curnt | Current |
| Curr | Currency | Cust | Customer |
| DB | Database | Dec | Decimal |
| Ded | Deduction | Del | Deleted |
| Dep | Deposit | Desc | Description |
| Dtl | Detail | Dflt | Default |
| Dfrd | Deferred | DirDep | Direct Deposit |
| Disc | Discount | Discont | Discontinued |
| Disp | Disposition | Dist | Distribution |
| DM | Debit Memo | Doc | Document |
| DS | Drop Ship | Dunn | Dunning |
| Earn | Earnings | Eff | Effective |
| Elim | Elimination | Emer | Emergency |
| Empl | Employee | Empr | Employer |
| Estab | Establish | Exch | Exchange |
| Excl | Exclude | Excpt | Exception |
| Exp | Expense | Expl | Explode |
| Ext | Extended:Extension:External | Fam | Family |
| Fed | Federal | Fin | Financial:Finance |
| Fisc | Fiscal | Fld | Field |
| Frt | Freight | Gtd | Guaranteed |
| HC | Home Currency | Hist | History |
| Hrs | Hours | ID | Identifier |
| Inc | Income | Incl | Include |
| Indus | Industry | Ins | Insurance |
| Int | Interest | Invc | Invoice |
| Invt | Inventory | Jrnl | Journal |
| Lang | Language | Liab | Liability |
| Loc | Location | Maint | Maintain |
| Max | Maximum | Meas | Measure |
| Meth | Method | Mgr | Manager |
| Min | Minimum | Mod | Modified |
| Msg | Message | MTD | Month to Date |
| Mth | Month | Mvmnt | Movement |
| Nat | Natural | NC | Natural Currency |
| Neg | Negative | No | Number |
| Nos | Numbers | Ofst | Offset |
| Opt | Option | Ord | Order |
| Orig | Original | Ovrd | Override |
| Pct | Percent | Pend | Pending |
| Per | Per:Period | Perm | Permission |
| Phys | Physical | Pmt | Payment |
| PP | Purchase Product | Post | Post:Posting |
| Prelim | Preliminary | Prev | Previous |
| Prim | Primary | Proc | Process |
| Prod | Product | Promo | Promotion |
| Pt | Point | Purch | Purchase |
| PTD | Period to Date | QTD | Quarter to Date |
| Pwd | Password | Qty | Quantity |
| Qtr | Quarter | Rcvd | Received |
| Rcpt | Receipt | Rcvr | Receiver |
| Rcvg | Receiving | Recur | Recurring |
| Recon | Reconcile | Reg | Registration |
| Ref | Reference | Remain | Remaining |
| Rel | Release | Req | Require |
| Reord | Reorder | Rev | Revised |
| Retnt | Retention | Revrs | Reverse |
| Reval | Revaluation | RGOL | Realized Gain or Loss |
| Rfnd | Refund | Rout | Routing |
| Rgstr | Register | Rplnsmnt | Replenishment |
| Rplcmnt | Replacement | Rsn | Reason |
| Rpt | Report | Rtrn | Return |
| Rstr | Restrict | Schd | Schedule |
| Rtrns | Returns | Seg | Segment |
| Secur | Security | Seq | Sequence |
| Sel | Selection | SP | Sales Product |
| Ship | Ship:Shipping:Shipment | Sper | Salesperson |
| Spec | Special | Srchg | Surcharge |
| Src | Source | Stax | Sales Tax |
| Stat | Status | Stmt | Statement |
| Std | Standard | Str | String |
| Stmts | Statements | Subj | Subject |
| Struct | Structure | Subsid | Subsidiary |
| Sub | Substitute:Submit | Susp | Suspense |
| Subst | Substitution | Tckt | Ticket |
| Sys | System | Tmplt | Template |
| Temp | Temporary | Tran | Transaction |
| Tot | Total | Trnslt | Translate |
| Trans | Transactions | Unappl | Unapplied |
| UGOL | Unrealized Gain or Loss | Vend | Vendor |
| Var | Variance | Vol | Volume |
| Ver | Version | Wght | Weight |
| Vouch | Voucher | YTD | Year to Date |
| Whse | Warehouse |

1. JavaScript
   1. Using Strict Mode

Strict Mode is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a “strict” operating context. This strict context prevents certain actions from being taken and throws more exceptions (generally providing the user with more information and a tapered-down coding experience).

Strict mode helps in a few ways:

* It catches some common coding bloopers, throwing exceptions.
* It prevents or throws errors when relatively “unsafe” actions are taken (such as gaining access to the global object).
* It disables features that are confusing or poorly thought out.

Good:

Place it within a function to turn on strict mode within that context.

function imStrict(){

"use strict";

// ... your code ...

}

* 1. Cache your objects

Boosting script performance is to cache your objects. Often times, code will repeatedly access a certain object, excessive calls to JavaScript objects can wear down the browser, not to mention your computer's memory.

The term "cache your object" means storing a repeatedly access object inside a user defined variable, and using that variable instead in subsequent references to the object. The performance improvement can be significant.

Bad:

batchUI.batchModel.Invoices()[0].AccountTaxTab.DistCode("");

batchUI.batchModel.Invoices()[0].AccountTaxTab.DistDescription("");

batchUI.batchModel.Invoices()[0].AccountTaxTab.GLAccountCode("");

batchUI.batchModel.Invoices()[0].AccountTaxTab.GLAccountDescription("");

batchUI.batchModel.Invoices()[0].AccountTaxTab.Amount("");

batchUI.batchModel.Invoices()[0].AccountTaxTab.Comment("");

batchUI.batchModel.Invoices()[0].AccountTaxTab.LineNumber("");

Good:

var accountTaxTab = batchUI.batchModel.Invoices()[0].AccountTaxTab;

accountTaxTab.DistCode("");

accountTaxTab.DistDescription("");

accountTaxTab.GLAccountCode("");

accountTaxTab.GLAccountDescription("");

accountTaxTab.Amount("");

accountTaxTab.Comment("");

accountTaxTab.LineNumber("");

* 1. Using Enum

Using enums instead of use hard-coded numbers makes code easy to understand and maintain.

Bad:

switch (sg.utls.toInt(val)) {

case 1:

…

case 2:

…

case 3:

…

}

Good:

var batchEnum = batchEnum || {};

batchEnum.DocumentType = {

Invoice: 1,

DebitNote: 2,

CreditNote: 3,

Interest: 4,

RetainageInvoice: 5,

RetainageDebitNote: 6,

RetainageCreditNote: 7

};

switch (sg.utls.toInt(val)) {

case batchEnum.DocumentType.Invoice:

…

case batchEnum.DocumentType.DebitNote:

…

case batchEnum.DocumentType.CreditNote:

…

}

* 1. Don’t Repeat Yourself (“DRY”)

JavaScript code should follow the Don’t Repeat Yourself (“DRY”) principle.

Bad:

if (batchFlow == 1 || batchFlow == 3) {

batchUI.Invoice.IsTrmCodDis(false);

batchUI.Invoice.IsDueDtDis(false);

batchUI.Invoice.IsDiscDtDis(false);

…

} else {

var documentType = sg.utls.toInt(batchUI.Invoice.DocumentType());

if (batchFlow == 5 && documentType == 1 || documentType > 4) {

batchUI.Invoice.IsTrmCodDis(false);

batchUI.Invoice.IsDueDtDis(false);

batchUI.Invoice.IsDiscDtDis(false)

…

} else {

batchUI.Invoice.IsTrmCodDis(true);

batchUI.Invoice.IsDueDtDis(true);

batchUI.Invoice.IsDiscDtDis(true);

…

}

Good:

if (batchFlow == 1 || batchFlow == 3) {

invoiceFileds(false);

} else {

var documentType = sg.utls.toInt(batchUI.Invoice.DocumentType());

if (batchFlow == 5 && documentType == 1 || documentType > 4) {

invoiceFileds(false);

} else {

invoiceFileds(true);

}

function invoiceFileds(enabled) {

uiInvoice.IsTrmCodDis(enabled);

uiInvoice.IsDueDtDis(enabled);

uiInvoice.IsDiscDtDis(enabled);

…

}

Bad:

initDatePicker: function () {

sg.utls.kndoUI.datePicker("txtBatchDate");

sg.utls.kndoUI.datePicker("txtDocumentDate");

sg.utls.kndoUI.datePicker("txtPostingDate");

sg.utls.kndoUI.datePicker("txtDueDate");

sg.utls.kndoUI.datePicker("txtDiscountDate");

sg.utls.kndoUI.datePicker("txtTaxReportingRateDate");

sg.utls.kndoUI.datePicker("txtAsOfDate");

…

},

Good:

initDatePicker: function () {

var kendoUi = sg.utls.kndoUI;

var fields = ["txtBatchDate", "txtDocumentDate", "txtPostingDate", "txtDueDate", "txtDiscountDate", "txtTaxReportingRateDate", "txtAsOfDate"];

$.each(fields, function (index, field) {

kendoUi.datePicker(field);

});

},

* 1. Using “===” and “!==” for primitive values compare

Strict equality checks (===) must be used in favor of abstract equality checks (==).

The only exception is when checking for undefined and null by way of null.

Bad:

if (FinderGridHelper.totalRecordsCount == 0)

Good:

if (FinderGridHelper.totalRecordsCount === 0)

* 1. Magic Numbers

Avoid using magic numbers, instead use constants or assigned variables

Bad:

if (model.ExportResponse.Status() === 2 || model.ExportResponse.Status() === 3)

{ //Error or Completed

Good:

// Declare Constant.

var ExportResponseStatus = { Error: 1, Completed: 2 };

if (model.ExportResponse.Status() === ExportResponseStatus.Error || model.ExportResponse.Status() === ExportResponseStatus.Completed)

{ //Error or Completed

* 1. Comment code appropriate

Insert a comment at the top of a file about what the file is responsible for. Also put comments on functions or properties that are not obvious, especially around logic that may be hard to understand.

Specifying if a function is a callback function also helps to clarify the code.

1. jQuery
   1. Selectors

All selectors are not equally efficient. Selectors in order from fastest to slowest:

1. ID selectors
2. Element selectors (form, input, and so on)
3. Class selectors
4. Pseudo and Attribute selectors (:visible, :hidden, [attribute=value], and so on)

ID and element are the fastest selectors, as they are backed by native DOM operations.

Avoid unnecessary loops. If possible, use the selector engine to address the elements that are needed. There are places where loops cannot be substituted, but try your best to optimize.

Bad:

$(‘#menu a.submenu’).each(function(){

$(this).doSomething().doSomethingElse();

});

Better:

$(‘#menu a.submenu’). doSomething().doSomethingElse();

Finding a DOM element by its ID is the fastest way, both in JavaScript and in jQuery.Whenever possible, you should always use the ID selector instead of using classes or tag names, or other ways.

Also, when finding an element by its ID with the $() function, there is no need to specify a context, which is slower than without.

Bad:

$( '#foo', this );

Better:

$( '#foo' );

* 1. DOM Insertion

Every DOM insertion is costly.

You can minimize DOM insertions by building HTML strings and using single append as late as possible. Use detach() if doing heavy interaction with a node, then re-insert it when done.

* 1. Caching

One of the most used features of jQuery is its ability to retrieve a DOM element by simply passing a reference to the $() function, be it an ID or class or whatever.

Every time we use $() function jQuery searches for the DOM element, and if the element is found it creates an object representing a clone of that element, with added capabilities. This is true even for elements we already found using this function, so calling this function for the same elements more than once is redundant.

Instead of calling the $() function multiple times for the same element, we should always keep a reference of the queried elements in a variable, and use it every time we need to access the same element.

Bad:

$( '#foo' ).hide();

$( '#foo' ).css( 'color', 'red' );

$( '#foo' ).show();

Better:

var $foo = $( '#foo' );

$foo.hide();

$foo.css( 'color', 'red' );

$foo.show();

* 1. Chaining

In the following example, three consecutive times a methods are called on the object $foo.

Instead of repeating each time the $foo variable, we can also use the chaining functionality jQuery offers.

Most jQuery methods returns the same object on which we called the method. The advantage is that this requires less code, is easier to write, and runs faster.

Bad:

$( '#foo' ).hide();

$( '#foo' ).css( 'color', 'red' );

$( '#foo' ).show();

Better:

$foo.hide().css( 'color', 'red' ).show();

Instead of calling the $() function multiple times for the same element, we should always keep a reference of the queried elements in a variable, and use it every time we need to access the same element.

* 1. Object Literals

Passing an object literal as a parameter for jQuery methods avoids the need to call the same method more than once.

Bad:

var $foo = $( '#foo' );

$foo.css( 'color', 'red' );

$foo.css( 'width', '200px' );

$foo.css( 'height', '200px' );

Instead, call the CSS method only one time by passing an object literal as a parameter.

Better:

var $foo = $( '#foo' );

$foo.css({

'color': 'red',

'width': '200px',

'height': '200px'

});

* 1. CSS

Instead of changing CSS values with jQuery, use classes to change the appearance of elements. This way you can keep all your CSS in your CSS files, specify different appearances for different classes, and use jQuery to add or remove classes to apply your CSS.

Specify different classes in the CSS:

.blue {

color: blue;

}

.red {

color: black;

}

Use jQuery to add or remove classes to objects. The following adds a 'blue' class to each 'p' element:

$( 'p' ).addClass( 'blue' );

If we want to change the text color to red, we remove the blue class and add the red one:

$( 'p' ).removeClass( 'blue' ).addClass( 'red' );

* 1. Avoid multiple $(document).ready() calls

Executing code only after the DOM has been fully created, or the page content has been fully loaded, is a good way of executing scripts. But it's unnecessary to repeat $(document).ready() or $(window).load() for each script you want to run.

Better to just use it only once, and put your code inside a function:

$( document ).ready( function () {

// all your code here

});

$( window ).load( function () {

// all your code here

});

* 1. this vs. $(this)

Use $(this) only when you need the jQuery methods; otherwise, use this when you only need a DOM element.

$('button').click(function() {

alert('Button clicked: ' + $(this).attr('id'));

});

// Efficient

$('button').click(function() {

alert('Button clicked: ' + this.id);

});

Some more common examples:

* Use this.checked instead of $(this).is(':checked')
* Use $.data(this, 'thing') instead of $(this).data('thing')
  1. Use Sub-queries

jQuery allows us to run additional selector operations on a wrapped set. This reduces performance overhead on subsequent selections since we already grabbed and stored the parent object in a local variable.

<ul id="traffic\_light">

<li><input type="radio" class="on" name="light" value="red" /> Red</li>

<li><input type="radio" class="off" name="light" value="yellow" /> Yellow</li>

<li><input type="radio" class="off" name="light" value="green" /> Green</li>

</ul>

For example, we can leverage sub-queries to grab the active and inactive lights and cache them for later manipulation.

var $traffic\_light = $('#traffic\_light'),

$active\_light = $traffic\_light.find('input.on'),

$inactive\_lights = $traffic\_light.find('input.off');

Tip: You can declare multiple local variables by separating them with commas. Save those bytes!

* 1. Leverage Event Delegation (a.k.a. Bubbling)

Every event (click, mouseover, and so on) in JavaScript “bubbles” up the DOM tree to parent elements. This is incredibly useful when we want many elements (nodes) to call the same function. Instead of binding an event listener function too many nodes—very inefficient—you can bind it once to their parent, and have it figure out which node triggered the event.

For example, say we are developing a large form with many inputs, and want to toggle a class name when selected.

Bad:

$('#myList li).bind('click', function(){

$(this).addClass('clicked');

// do stuff

});

Instead, we should listen for the click event at the parent level.

Better:

$('#myList).bind('click', function(e){

var target = e.target, // e.target grabs the node that triggered the event.

$target = $(target); // wraps the node in a jQuery object

if (target.nodeName === 'LI') {

$target.addClass('clicked');

// do stuff

}

});

* 1. AJAX Methods

The various AJAX methods available to us might come across as daunting; though they needn't. In fact, most of them are simply helper methods, which route directly to $.ajax.

As an example, let's review getJSON, which allows us to fetch JSON

$.getJSON('path/to/json', function(results) {

// callback

// results contains the returned data object

});

Behind the scenes, this method first calls $.get

getJSON: function( url, data, callback ) {

return jQuery.get(url, data, callback, "json");

}

$.get then compiles the passed data, and, again, calls the "master" (of sorts) $.ajax method.

get: function( url, data, callback, type ) {

// shift arguments if data argument was omited

if ( jQuery.isFunction( data ) ) {

type = type || callback;

callback = data;

data = null;

}

return jQuery.ajax({

type: "GET",

url: url,

data: data,

success: callback,

dataType: type

});

}

Finally, $.ajax performs a massive amount of work to allow us the ability to successfully make asynchronous requests across all browsers. You can use the $.ajax method directly and exclusively for all your AJAX requests. The other methods are simply helper methods.

* 1. Passing an Attribute Object

As of jQuery 1.4, we can now pass an object as the second parameter of the jQuery function. This is helpful when we need to insert new elements into the DOM. For example:

Before:

$('#foo')

.attr({

id : 'someId',

className : 'someClass',

href : 'somePath.html'

});

After:

$('#foo', {

id : 'someId',

className : 'someClass',

href : 'somePath.html'

});

Not only does this save a few characters, but it also makes for cleaner code. In addition to element attributes, we can even pass jQuery specific attributes and events, like click or text.

* 1. Use live()

Attach an event handler for all elements which match the current selector, now and in the future.

$(selector).live(events, data, handler); // jQuery 1.3+

$(document).delegate(selector, events, data, handler); // jQuery 1.4.3+

$(document).on(events, selector, data, handler); // jQuery 1.7+

* 1. Use jQuery.noConflict()

Many JavaScript libraries use $ as a function or variable name, just as jQuery does. In jQuery's case, $ is just an alias for jQuery, so all functionality is available without using $.

If we need to use another JavaScript library alongside jQuery, we can return control of $ back to the other library with a call to $.noConflict()

<script type="text/javascript" src="other\_lib.js"></script>

<script type="text/javascript" src="jquery.js"></script>

<script type="text/javascript">

$.noConflict();

// Code that uses another library's $ can follow here.

</script>