|  |
| --- |
| Coding Best Practices |
| ELCA VN |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Imput. | Report | Version | Date | Author(s) | Status | Visa |
| 201-604 |  | 1.13 | 26.05.14 | PHD, HNT | Valid | QKP |
| @ ELCA Information Technology (Vietnam) Ltd. | | | | | | |

I. Table of Contents

I. Table of Contents 2

II. Record of changes 3

III. References 3

IV. Abbreviations 4

1 Introduction 5

1.1 Purpose 5

1.2 Sources of coding best practices 5

1.3 How to use the document 5

1.3.1 Reading for the first time 5

1.3.2 Use as a reference 6

2 General programming 7

2.1 BP201: Prefer for-each to traditional loops 7

2.2 BP202: Do not use Assertion at all in the code except for unit tests 7

2.3 BP203: Be careful of “TODO” leftovers in code 8

2.4 BP204: Comparing two objects (“.equals” instead of “==”) 10

2.5 BP205: Always invoke equals() on constants when comparing them with variables 11

2.6 BP206: Minimize the scope of local variables 12

3 Exception handling 13

3.1 BP301: Never leave empty catch block 13

3.2 BP302: Be specific when throwing exception 14

3.3 BP303: Include information about failure when throwing exception 15

3.4 BP304: Always put try..catch in finally (if needed) 18

3.5 BP305: Always log exception with appropriate severity 19

4 Resources management 21

4.1 BP401: All resources must be closed after used 22

4.2 BP402: Don’t do lazy initialization unless you need to 24

4.3 BP403: Use double-check idiom when lazy-initializing object in threaded context 25

5 Design 27

5.1 BP501: Never store information in stateless components 27

5.2 BP502: Serializable should not be implemented for all data objects 27

5.3 BP503: Mark classes, methods as final if they should not be extended/overridden 28

5.4 BP504: Prefer empty list, array over null object when designing API with collection returning type 30

5.5 BP505: The constants and configurable parameters are different 32

5.6 BP506: Use enums instead of integer constants 32

6 Transaction management 35

6.1 BP601: Statements modifying database structure (DDL) are auto-committed 35

6.2 BP602: Single unit-of-work (use-case) should be enclosed inside one database transaction to preserve data integrity 36

7 Performance 39

7.1 BP701: Limit remote invocations between application layers 39

7.2 BP702: Beware the performance of string concatenation 41

8 Security 42

8.1 BP801: Avoid SQL Injection by using parameterized queries 42

8.2 BP802: Do not build HTML by concatenating strings. 43

9 ORM 45

9.1 BP901: Avoid N + 1 selects 45

9.2 BP902: Load only necessary data from database 47

1. Record of changes

**Template**

|  |  |  |  |
| --- | --- | --- | --- |
| **Filename** | **Version** | **Date** | **Description / Author** |
| S07-ProjectHistory-01.doc | 0.1 | 12.04.05 | Initial version / JPO |
| S07-ProjectHistory-02.doc | 0.2 | 16.09.05 | Corrections after review / JPO |
| Coding best practices\_14.doc | 1.4 | 20.04.10 | Add 3 BP (from COIN recurrent defects analysis 06.2009):  BP205, BP520, BP703  Fixed BP103, BP302  Cleaned document layout  /JPO |
| Coding best practices\_15.doc | 1.5 | 07.05.10 | Corrections after DR multiple /JPO |
| Coding best practices\_16.doc | 1.6 | 05.10.10 | Added introduction / QAN, JPO |
| Coding best practices\_17.doc | 1.7 | 10.04.12 | Update according to result of Management Workshop in 10.11.11 / PHD |
| Coding Best Practices\_18.doc | 1.8 | 10.04.12 | Updates for C# on BP402, BP404, BP502, BP503 / PHD |
| Coding Best Practices\_19.doc | 1.9 | 28.12.12 | Update according to OFI#249 / PHD |
| Coding Best Practices\_1.10.doc | 1.10 | 01.04.13 | Update according to “Recurrent Issues Analysis” of 2012 / PHD |
| Coding Best Practices\_1.11.doc | 1.11 | 15.04.13 | Update SQL Injection sample / HNT |
| Coding Best Practices\_1.12.doc | 1.12 | 09.10.13 | Update all items’ severity to synchronize with JIRA’s definition (OFI249) / PHD |
| Coding Best Practices\_1.13.docx | 1.13 | 26.05.14 | Add chapters on exception logging and HTML string concatenation / HNT |

III. References

|  |  |
| --- | --- |
| [Joshua08] | Joshua Bloch. *Effective Java – Second Edition*. Addison-Wesley, 2008 |
|  |  |

IV. Abbreviations

|  |  |
| --- | --- |
| DB | Database |
| DDL | Data Definition Language |
| DTO | Data Transfer Object |
| OOP | Object Oriented Programming |
| PROD | Production |
| ORM | Object Relational Mapping |

# Introduction

This document contains many best practices used in coding activities at ELCA VN. Best practices are broken down in various categories:

* General programming
* Exception handling
* Resources management
* Design
* Transaction management
* Performance
* Security

All engineers at ELCA are required to know this document and apply its content in their daily coding tasks in projects.

## Purpose

Coding best practices are practices that, when understood and applied, ensure:

* Reduction of code defects (of various kinds and severity levels) by showing common coding traps and mistakes and ways to avoid them
* Increase of code readability and maintainability by providing a common and consistent way to code across team members and even across projects.

Understanding and applying these coding best practices will help every engineer become a better programmer.

## Sources of coding best practices

The content of this document is mainly sourced from results of recurrent defects analysis performed by ELCA Code Inspection teams. This is the main source for adding new Best Practices to this document.

Another source is contractor feedback on code created by ELCA VN engineers.

Finally, comments, provided by ELCA VN engineers themselves, based on their effective experience in projects, also help ameliorate this document by improving existing Best Practices.

## How to use the document

### Reading for the first time

If you are a newcomer, you have to read this document during the first month trial. There are many best practices in the document to read; hence you are advised to read three best practices every day. Thus, you can cover all practices by the end of the first month. Best Practices are not supposed to be learned “by heart”.

**Support from your coach**

You receive a training schedule for the first month trial from HR, indicating which best practices to read each day. Follow the plan and note any unclear points; these points will be clarified by your coach during short meetings every week.

### Use as a reference

When you already have read this document, you can refer to it at any time during coding when you face a problem which you remember is addressed in one Best Practice (you can quickly look up best practices in the Table of Contents).

# General programming

## BP201: Prefer for-each to traditional loops

|  |  |
| --- | --- |
| **Problem** | Traditional loops are error-prone for iterating over collections |
| **Type** | Reliability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | Once being used in traditional loops, the iterator and index variable occur many times in each loop, which resulting in chances to get them wrong.  The for-each loop gets rid of the opportunity for error by hiding the iterator or index variable completely. Its advantages over the traditional become greater when there are nested iterations over multiple collections. |
| **Good code / behavior** |  |

## BP202: Do not use Assertion at all in the code except for unit tests

|  |  |
| --- | --- |
| **Problem** | Assertion is not designed for business checking |
| **Type** | Reliability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | Assertion should not be used in place of business checking because of the following reasons:  - it is means for programming error check and not for business code  - it can be turned off and cause the validation to fail  - it cannot throw specific Exception |
| **Good code / behavior** |  |
| **Comment** | This is true for public methods, but for private, you can still use assertions (because that would be programming errors here) |

## BP203: Be careful of “TODO” leftovers in code

|  |  |
| --- | --- |
| **Problem** | Nobody knows what “to do” with TODO comments left in code |
| **Type** | Maintainability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | Many times, developers keep auto-generated TODO comments in their code even though they have filled in the matching code block adequately.  Also, sometimes, developers add some TODO comments just to remember that they should review some code part later on; however, very often, those TODO comments are never performed by the author and are kept for a long time. Later on, when they are discovered, nobody remembers what to do because those comments are not clear enough about the task to be performed and the necessary conditions to perform that task. |
| **Good code / behavior** | Every **TODO** comment should clearly specify the following information:   * **Who**: the visa of the author of this TODO comment (and probably the one who will have to perform this task later on) * **When**: the latest date (or milestone) at which this task should be performed in the code; this must also include conditions (if any) for this task to be start-able (e.g. external constraint, library version change…) * **Why**: the reason why we plan a change in the code and it can’t be done right now; this can help, later on, figuring out how important this task is for the project. * **What**: a good (detailed) description of the task that must be performed (so that it can be performed by someone else than the comment’s author if needed). This could include important advice (or warning) if there is some risk changing that code. |

## BP204: Comparing two objects (“.equals” instead of “==”)

|  |  |
| --- | --- |
| **Problem** | Operator “**==**” actually compares the two object references and may result in wrong behavior |
| **Type** | Functionality |
| **Severity** | Serious |
| **Bad code / behavior (Java)** |  |
| **Description** | When using operator “**==**”, the two involved object references are compared to see if they point to the same memory location.  In Java, we cannot compare, for example, two instances of **java.lang.String** for equality with “**==**”. We must instead use the **equals()** method, which is inherited by all classes from **java.lang.Object**.  In C#, the same principle hold, **except** for strings themselves, where “**==**” is equivalent to “**Equals**”. |
| **Good code / behavior (Java)** |  |

## BP205: Always invoke equals() on constants when comparing them with variables

|  |  |
| --- | --- |
| **Problem** | NullPointerException may occur if equals() is invoked on variables when comparing them with constants |
| **Type** | Reliability |
| **Severity** | Non critical |
| **Bad code / behavior** |  |
| **Description** | NullPointerException will occur in method initAction() above if txtFirstName.getText() returns null. Thus, it is a good practice to compare strings as shown below, that doesn’t result in an exception if the variable is indeed null. |
| **Good code / behavior** |  |

## BP206: Minimize the scope of local variables

|  |  |
| --- | --- |
| **Problem** | Extending local variables’ scope reduces readability, maintainability and increase the likelihood of error |
| **Type** | Maintainability |
| **Severity** | Non critical |
| **Bad code / behavior** | N/A |
| **Description** | Declaring a local variable prematurely can cause its scope not only to extend too early, but also to end too late. The scope of a local variable extends from the point where it is declared to the end of the enclosing block. If a variable is declared outside of the block in which it is used, it remains visible after the program exits that block. If a variable is used accidentally before or after its region of intended use, the consequences can be disastrous.  Thus, it is advised to declare the variable just before used. By minimizing the scope of local variables, you increase the readability and maintainability of your code and reduce the likelihood of error. |
| **Good code / behavior** | N/A |

# Exception handling

## BP301: Never leave empty catch block

|  |  |
| --- | --- |
| **Problem** | Ignoring an exception will result in no way to trace back to the cause of the problem once it occurs |
| **Type** | Reliability |
| **Severity** | Critical |
| **Bad code / behavior** |  |
| **Description** | Ignoring an exception is analogous to ignoring a fire-alarm and will result in:  - an application silently continues in case of error  - no easy way to trace back to original source of error  In normal case (first case in the example above), it is advised to:  - log the exception in “error” level to ensure that the exception is traced back in case client code forgets to do that  - propagate the original, or respectively translate and then propagate the translated exception outward to help preserving information to aid in debugging the failure  At the very least (second case in the example above), the catch block should contain a log of the exception to help investigating the matter if the exception happens more often. |
| **Good code / behavior** |  |

## BP302: Be specific when throwing exception

|  |  |
| --- | --- |
| **Problem** | It is difficult for client code to deal properly with “*generic*” exceptions |
| **Type** | Reliability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | In above example, it is difficult for client code to figure out which specific error has occurred |
| **Good code / behavior** |  |

## BP303: Include information about failure when throwing exception

|  |  |
| --- | --- |
| **Problem** | Lacking of knowledge about the failure may create difficulty in handling it |
| **Type** | Reliability |
| **Severity** | Serious |
| **Bad code / behavior** | **Exception definition:**  **Service code:**  **Presentation code:** |
| **Description** | In above example, it’s difficult for client code to figure out what action should be carried out next, since there is no or only a little hint from the exception |
| **Good code / behavior** | **Exception definition:**    **Service code:**  **Presentation code:** |

## BP304: Always put try..catch in finally (if needed)

|  |  |
| --- | --- |
| **Problem** | Logging information of unusual error may be lost if try..catch is forgotten in finally |
| **Type** | Reliability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | If methods invoked from within finally clause can themselves throw exceptions, they must be wrapped inside a try..catch block with adequate logging. This will help to log and analyze any problem that happens when closing the resource (although this is unusual). |
| **Good code / behavior** |  |

## BP305: Always log exception with appropriate severity

|  |  |
| --- | --- |
| **Problem** | Important logging information can be lost when system log level is changed. |
| **Type** | Reliability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | Although the above exception stacktrace is required for determine issues with the application, it is logged with DEBUG level. This log level is unlikely to be used in productive systems; therefore the information can be lost forever. |
| **Good code / behavior** | Always log the information with its respective severity. It’s always good to check if the specified level is enabled or not to reduce the computational cost to construct log messages. |

# Resources management

## BP401: All resources must be closed after used

|  |  |
| --- | --- |
| **Problem** | Forgetting to properly close a resource after used will cause it to be exhausted |
| **Type** | Efficiency |
| **Severity** | Critical |
| **Bad code / behavior** |  |
| **Description** | Expensive resources must always be cleaned up explicitly, by invoking the **close()** method from the **finally** block that surrounding its initializing place. Otherwise, these resources will soon be exhausted and cause serious consequences to the functionality/availability/scalability of the application.  Here is a list of some resources that are considered expensive:   * Streams (both input and output) * Readers * Writers * Database connections * SQL statements |
| **Good code / behavior (Java)** | NOTE:  - In C#, “using” keyword can be used to replace try ... finally to dispose resource.  - In Java 7+, the try-with-resources statement could be used to close the resources regardless of whether the try statement completes normally or abruptly: http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html |
| **Comment** | The following approach which delegates the closing task to a helper should be avoided as some of the available libraries (like org.apache.commons.io.IOUtils) just silently ignore any exception might happen. |

## BP402: Don’t do lazy initialization unless you need to

|  |  |
| --- | --- |
| **Problem** | Lazy initialization is used for optimizations only and becomes tricky in the presence of multiple threads |
| **Type** | Efficiency |
| **Severity** | Critical |
| **Bad code / behavior** |  |
| **Description** | Lazy initialization is the act of delaying the initialization of a field until its value is needed. It is a double-edged sword while decreasing the cost of initializing a class or creating an instance, at the expense of increasing the cost of accessing the lazily initialized field.  Especially, if two or more threads share a lazily initialized field, it is critical that some form of synchronization be employed, or severe bugs can result.  Thus, under most circumstances, normal initialization is preferable to lazy initialization. |
| **Good code / behavior** |  |
| **Comment** | When lazy initialization of an instance variable is unavoidable, double-checked-locking mechanism must be used (see the “***Use double-check idiom when lazy-initializing object in threaded context***” best practice) |

## BP403: Use double-check idiom when lazy-initializing object in threaded context

|  |  |  |
| --- | --- | --- |
| **Problem** | If two or more threads share a lazily initialized field, it is critical that some form of synchronization be employed, or severe bugs can result | |
| **Type** | Efficiency | |
| **Severity** | Critical |
| **Bad code / behavior** |  | |
| **Description** | Supposed more than one thread arrive at the getEmployeeCache() method. All the threads finds the employeeCache to be null and they will respectively take turn to initialize the expensive cache, which is not desired.  A better way is to check employeeCache again in locking mode to see if some other thread has already initialized it. | |
| **Good code / behavior** | In Java:    In .NET, see <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/ImpSingletonInCsharp.asp> | |
| **Pitfall** | Double-check idiom does not work reliably with all JVM (1.5+ only)! But at minimum, should do the “single checked locking” (i.e. put “synchronized” at method declaration) | |

# Design

## BP501: Never store information in stateless components

|  |  |
| --- | --- |
| **Problem** | Storing information in stateless components will make them become stateful and cause serious problems in multi thread environment |
| **Type** | Functionality |
| **Severity** | Critical |
| **Bad code / behavior** |  |
| **Description** | As EmployeeServiceImpl is a stateless service and used in a multi thread environment, there is chance that multiple threads access method update() at the same time and modify value of the shared private attribute defaultCommision that is being read by other threads. |
| **Good code / behavior** |  |

## BP502: Serializable should not be implemented for all data objects

|  |  |
| --- | --- |
| **Problem** | Implementing Serializable comes with costs and should be avoided as much as possible |
| **Type** | Maintainability |
| **Severity** | Non critical |
| **Bad code / behavior** | Implementing Serializable for all data objects |
| **Description** | Implementing Serializable will:  - decreases the flexibility to change a class’s implementation once it has been released because its byte-stream encoding (or serialized form) becomes part of its exported API  - increases the likelihood of bugs and security holes because relying on the default deserialization mechanism can easily leave objects open to invariant corruption and illegal access  - increases the testing burden associated with releasing a new version of a class because it is mandatory to check that it is possible to serialize an instance in the new release and deserialize it in old releases, and vice versa  Thus, it is advised to implement the Serializable only for serializable data objects |
| **Good code / behavior** | Only apply it for data objects that need to be serialized:  + to transfer via network (.e.g. RMI)  + to save to disk by cache libraries (e.g. ehCache)  + to store in HTTPSession  …  NOTE: ISerializable interface and Serializable attribute are used in C# for serialization |

## BP503: Mark classes, methods as final if they should not be extended/overridden

|  |  |
| --- | --- |
| **Problem** | Classes/methods whose behaviors are fixed but could still be technically extended/overridden will lead to unexpected and wrong behaviors |
| **Type** | Maintainability |
| **Severity** | Serious |
| **Bad code / behavior (Java)** |  |
| **Description** | Method doImport() might be mistakenly changed in subclasses and cause undesired effect, e.g. forgetting to send notification after the import finishes. |
| **Good code / behavior (Java)** | NOTE: In C#, use "sealed" keyword instead of "final". Besides, non-static methods in C# are sealed by default (different than in Java). |
| **Pitfall** | Marking a method/class as final hinders extensibility. In libraries or frameworks, be very careful because not all cases might have been foreseen. If it happens, it causes a lot of troubles since the final classes or methods cannot be extended. |

## BP504: Prefer empty list, array over null object when designing API with collection returning type

|  |  |
| --- | --- |
| **Problem** | Returning NULL object for collection type always makes client code unnecessarily complex |
| **Type** | Maintainability |
| **Severity** | Serious |
| **Bad code / behavior** | **Service code:**    **Presentation code:** |
| **Description** | Returning NULL object for collection type always makes client code unnecessarily complex as a nullify check must to be applied before working with the result |
| **Good code / behavior** | **Service code:**  **Presentation code:** |

## BP505: The constants and configurable parameters are different

|  |  |
| --- | --- |
| **Problem** | Although they are different concepts, constants are mistakenly used as configurable parameters and vice versa |
| **Type** | Maintainability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | If a value will be never changed, or if it changes, we have to rebuild the system. Then put it as constants.  Else make it as configurable parameters, so that it can be configured when the system is deployed in different environment (development, test, production, .etc.). |
| **Good code / behavior** |  |

## BP506: Use enums instead of integer constants

|  |  |
| --- | --- |
| **Problem** | Integer constants are used to represent object’s internal states |
| **Type** | Maintainability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | Whenever you find yourself writing the above code to represent internal states or flags of a module, consider using enums instead. Enums are a better approach than integer constants because they are forced to fit within the range of allowed values, whereas using an int variable can contain the whole range of integers even if only a few of them are valid states.  Also, enums ease debugging and logging because you can print the human readable value by directly invoking the method toString(), while when using integers you would have to write a special purpose function to provide this functionality. |
| **Good code / behavior** |  |

# Transaction management

## BP601: Statements modifying database structure (DDL) are auto-committed

|  |  |
| --- | --- |
| **Problem** | Statements modifying database structure (DDL) are auto-committed |
| **Type** | Reliability |
| **Severity** | Critical |
| **Bad code / behavior** | **Service code:**    **Batch code:** |
| **Description** | The configuration to rollback transaction in case of errors in the example above will **not** work as the transaction itself has been already committed before when either disableDatabaseIndexes() or enableDatabaseIndexes() which contains DDL statements is called. |
| **Good code / behavior** | **Service code:**    **Batch code:** |
| **Comment** | Using DDL in a program is not really a good practice! Try to avoid this. |

## BP602: Single unit-of-work (use-case) should be enclosed inside one database transaction to preserve data integrity

|  |  |
| --- | --- |
| **Problem** | Splitting single unit-of-work(use-case) into multiple database transactions can easily lead to data integrity violation |
| **Type** | Reliability |
| **Severity** | Critical |
| **Bad code / behavior** | **Presentation code:**    **Service code:** |
| **Description** | Because EmployeeServiceImpl and DepartmentServiceImpl are configured to be transactional (every call from outside is wrapped inside a dedicated transaction), the single use-case promoteToManager(…) is split into 2 database transactions; hence, might leave database in an inconsistent state if error occurs within departmentService.save(…). Consequently, there will be a department whose manager is NOT yet a manager in such case.  Good approach is to encapsulate all database modifications derived from single unit-of-work (use-case) into one dedicated database transaction as example below. With this, the whole promoteToManager(…) method can either fully succeed or fail without leaving any inconsistency in database, thanks to the all-or-nothing “*atomicity*” of database transaction. |
| **Good code / behavior** | **Presentation code:**    **Service code:** |
| **Comment** | It is recommended to:   * have transaction demarcation (enclosing of sequential actions in transactional behavior) at service layer because it should be driven by business logic * have transaction demarcation declared at class level (instead of method) with propagation mode set to REQUIRED (support a current transaction, create a new one if none exists) for better control over transactional behavior and maintainability   In case another propagation mode is required:   * create a dedicated technical service without any business logic inside * strongly emphasize the technical purpose in its name (i.e. AlwaysExecuteInNewTransactionService) * delegate all the work to “*real*” business services * follow the second recommendation above to configure it with desired propagation mode accordingly |

# Performance

## BP701: Limit remote invocations between application layers

|  |  |
| --- | --- |
| **Problem** | Too many remote invocations between application layers |
| **Type** | Efficiency |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | When working with remote interfaces, every call is expensive. As a result, we need to reduce the number of calls, and that means to transfer more data within each call. The solution is to create DTO(s) that can hold all data for the call. It needs to be serializable to go across the connection. Usually an assembling mechanism is used on the server side to convert data between the DTO(s) and domain objects. |
| **Good code / behavior** |  |

## BP702: Beware the performance of string concatenation

|  |  |
| --- | --- |
| **Problem** | String concatenation performs horribly if the number of items is large |
| **Type** | Efficiency |
| **Severity** | Non critical |
| **Bad code / behavior** |  |
| **Description** | Operation on String (immutable) objects creates a lot of short-live objects. As consequence, using the string concatenation operator repeatedly to concatenate ***n*** strings requires time quadratic in ***n***.  To achieve acceptable performance, use a StringBuilder in place of String to store the under-construction string. |
| **Good code / behavior** |  |

# Security

## BP801: Avoid SQL Injection by using parameterized queries

|  |  |
| --- | --- |
| **Problem** | SQL injection |
| **Type** | Functionality |
| **Severity** | Critical |
| **Bad code / behavior** |  |
| **Description** | SQL injection is a [code injection](http://en.wikipedia.org/wiki/Code_injection) technique that exploits a [security vulnerability](http://en.wikipedia.org/wiki/Security_vulnerability) occurring in the [database](http://en.wikipedia.org/wiki/Database) layer of an [application](http://en.wikipedia.org/wiki/Application_software). The vulnerability is present when user input is either incorrectly filtered for [string literal](http://en.wikipedia.org/wiki/String_literal) [escape characters](http://en.wikipedia.org/wiki/Escape_sequences) embedded in [SQL](http://en.wikipedia.org/wiki/SQL) statements or user input is not [strongly typed](http://en.wikipedia.org/wiki/Strongly-typed_programming_language) and thereby unexpectedly executed.  In the above example, it is possible for attackers to provide a username containing SQL meta-characters that subvert the intended function of the SQL statement. For instance, by providing a username of  admin' OR '1'='1’--  and a blank password, the generated SQL statement becomes:  This allows attackers to log in to the site without supplying a password, since the ‘OR’ expression is always true. Using the same technique attackers can inject other SQL commands which could extract, modify or delete data within the database.  Thus, it is recommended to always use parameterized queries so that variables passed as arguments to the queries will automatically be escaped by the database connection driver. |
| **Good code / behavior** |  |

## BP802: Do not build HTML by concatenating strings.

|  |  |
| --- | --- |
| **Problem** | HTML elements are dynamically built by string concatenation in JavaScript |
| **Type** | Reliability |
| **Severity** | Serious |
| **Bad code / behavior** |  |
| **Description** | There are multiple issues with the code above:   * If the refData[j].Children[k].Url or refData[j].Children[k].DisplayName contains the single quote (‘), the code is likely to break * The refData[j].Children[k].Url can be injected with malicious code and the page is vulnerable to XSS attacks. * The code can grow complex easily and it’s difficult to maintain later.   Therefore, it is recommended that :   * Always escape the inputs before using them for building HTML * Templating frameworks should be used for dynamically constructing HTML elements on client side in most of the cases. |
| **Good code / behavior** | Example with jQuery template |

# ORM

## BP901: Avoid N + 1 selects

|  |  |
| --- | --- |
| **Problem** | N + 1 selects |
| **Type** | Efficiency |
| **Severity** | Critical |
| **Bad code / behavior** | **Hibernate mapping:**    **Service-layer code:**    **Generated SQL:** |
| **Description** | As Department has a lazy OneToMany relationship to Employee, every access on employees attribute of Department results in a SQL query to populate the corresponding list.  Suppose that N departments are return by the first query, there will be in total (N + 1) queries generated by this code, including:  - 1 query to get all departments  - N queries to get all employees for each department  This always has a very bad impact on performance as too many queries and roundtrips between application server and database are generated. |
| **Good code / behavior** | **1st option: Early-fetching all necessary associations:**  **JAVA CODE:**    **Generated SQL:**    **2nd option: Moving server operation to database (if applicable):**  **JAVA CODE:**    **SQL STATEMENT:** |

## BP902: Load only necessary data from database

|  |  |
| --- | --- |
| **Problem** | Loading the whole information from database and then, applying filter in memory can cause very poor performance with big data |
| **Type** | Efficiency |
| **Severity** | Critical |
| **Bad code / behavior** | **SHAREPOINT CODE:** |
| **Description** | SPList.Items.GetItemByid does actually 2 steps:   * Loading the entire SPList.Items collection into memory * Filtering the previously loaded list by ID   This always causes poor performance for big collection of SPList.Items as there are too much data loaded from database and sent over the network to application server. |
| **Good code / behavior** | **Using the right API to apply filtering on database so that only the necessary data is loaded and sent over network**  **SHAREPOINT CODE:** |
| **Comment** | In other ORM (i.e. Hibernate/NHibernate/EntityFramework), there are typically 2 techniques used to limit the data queried from database:   * Apply projection to load only required columns from database (instead of the whole entity) 🡺 This is recommended for search screens as the number of displayable columns are usually much more smaller than the number of those in database. * Apply pagination to load only a limited number of rows from database (instead of all matching rows) 🡺 This is recommended for:   + first matching row search as only the first row is accessed by database and then, sent to application server   + pagable screens as loading the whole data and then, paging in memory is rarely a good solution in term of performance   + batch processing as loading a huge amount of data can slow down or even kill the application server |