Capgemini

7/3/2019

CAPGEMINI

WELCOME PACKAGE

CODING STANDARDS

Revision History:

Version	Date	Author	Reviewed By	Revision Notes
1.0	07/03/2019	Sandesh Chauhan		Initial draft

TABLE OF CONTENTS

Section 1. Purpose of Document

This document describes a collection of standards, conventions, and guidelines for writing solid Java code. They are based on sound, proven software engineering principles that lead to code that is easy to understand, to maintain, and to enhance.

Furthermore, thsis document summarizes the various coding standards followed in the project carried on for migration of welcome package project from spring mvc to spring boot environment.

Section 2. Scope

The document first gives a glance of the general coding standards followed across globe to develop a clean application. Later , the document puts focus on welcome package specific features.

Section 3. Coding Styles & Techniques

The following section gives a brief of what best practices should be adhered in order to develop and maintain a clean, fast and modularized software application.

3.1. Code Organization

Classes should be organized as follows;

- Java Docs
- Package Identifier
- Class JavaDoc
- Class Declaration
- Log4J Logger Declaration
- Constants
- Member Variables
- Constructors
- Member Functions

3.2. Naming

One of the key tenets of good design is having a clear understanding of the responsibility of packages, classes and methods. This should be clearly understood by the reader through the naming of packages, classes and methods.

The process of choosing good names challenges the core understanding of responsibilities in code and design.

The following are general naming convention guidelines :

- Use full English descriptors that accurately describe the variable, field, and class; for example, use names like firstName, grandTotal, or CorporateCustomer.
- Use terminology applicable to the domain.

- Use mixed case to make names readable. Use lowercase letters in general, but capitalize the first letter of class names and interface names, as well as the first letter of any non-initial word.
- Use abbreviations sparingly, but if you do so then use them intelligently. A list of standard short forms (abbreviations) should be maintained.
- Avoid long names.
- Avoid names that are similar or differ only in case.
- Avoid (leading or trailing) underscores.
- All member data will begin with a lowercase letter, and then follow the normal naming convention for other identifiers (mixed case, each word beginning with uppercase).
- Parameter names should be constructed like identifiers, do not include the type in the name.
- Method variable names (not class member variables) should be constructed like identifiers, the exceptions are the commonly used i, j ,k etc for for loop variables.

3.3. METHODS SHOULD HAVE ONE EXIT POINT

Code with multiple returns can be hard to read / understand. This also tends to be a symptom of large methods thats should be refactored down. Usage od continue statement should be avoided.

3.4. HASHCODE(), EQUALS() AND TOSTRING()

All the entity classes should by default implement/override hashcode() ,equals() and toString() method.Use equals() instead of == when comparing objects. If you use == you are comparing an objects id's not its values. If someone defined an equals method to compare objects, then they want you to use it. Otherwise, the default implementation of Object.equals is just to use ==. Example . Whenever a.equals(b), then a.hashCode() must be same as b.hashCode().

3.5. LOGGER : LOG MESSAGES , LEVELS

Always check the logging level before issuing a log statement if the statement being logged contains any computation e.g. concatenation. This reduces the impact of lower level messages when a higher logging level is applicable.

Based of the framework the log4j logging levels may differ. Decide the logging levels based on the event occurred or may occur.

Example :

FATAL – server failed, operations have to do something immediately to stand it back up (if the system runs unattended at some times, this would – through OpenView or Tivoly – alert the on-call operator)
ERROR – data loss, but can proceed
WARN – no data loss, but something fishy (e.g. inconsistent data received on an interface – no immediate concern, but someone should look into this)
INFO – regular events, e.g. batch has completed, server startup completed, shutdown initiated etc.
DEBUG – debugging information by the developer, can stay activated during a performance test
TRACE – very detailed information, should be disabled during performance tests (would be active during unit tests etc.)

3.6. SEMANTICS

The names , levels ,status ,comments given to a particular entity or code snipet should make sense and should be relevant to the task it is expected to perform of represent. It should be meaningful. Avoid using non-intuitive language constructs just to reduce the amount of code.

3.7. IMPORTS

Avoid * forms of import. Be precise about what you are importing so as to avoid subtle bugs where you think you are using classes of the same name but that live in different packages.
Check that all declared imports are actually used (a good IDE will highlight those not used). This makes it easier for readers of your code to understanding its context and dependencies.
Use your IDE's 'add import' and 'organise imports' commands to make this easier.

3.8. INITIALIZATION OF VARIABLES

Declare a local variable only at that point in the code where its initial value is known. The only exceptions to this rule are loop variables which should be declared outside the loop for efficiency reasons and return values that should be declared at the beginning of the method

Declare and initialize a new local variable rather than reusing (reassigning) an existing one whose value happens to no longer be used at that program point. Especially loop variables.

3.9. COMMENTS

Types of comments

Java has three styles of comments:

- documentation comments that start with /** and end with */
- C-style comments that start with /* and end with */
- single-line comments that start with // and go until the end of the source-code line.

Usage

Use documentation comments immediately before declarations of interfaces, classes, member functions, and fields to document them. Documentation comments are processed by javadoc, see below, to create external documentation for a class.

Use single line comments internally within member functions to document business logic, sections of code, and declarations of temporary variables.

Javadoc processes Java code files and produces external documentation, in the form of HTML files, for your Java programs. Javadoc supports a limited number of tags; reserved words that mark the beginning of a documentation section.

Example

Documentation : /**Customer: A customer is any
 *person or organization that we
 *sell services and products to.
 @author S.W. Ambler*/

Single line : // Apply a 5% discount to all invoices
//over $1000 due to generosity
//campaign started in Feb. of 1995.

Mark unfinished code with an @TODO annotation including a description. You should generally avoid checking in code with outstanding // TODO comments in it - this means you're not finished. Change then to @TODO instead.

3.10. Declarations

- Classes should be declared as follows using capitalized English words for the class name.
- Each class should declare a logback/slf4j logger instance.
- The Logger declaration may be omitted for an Abstract class that does no logging.
- A default (no args) constructor may be omitted, as the compiler will define one.
- Member variables should be declared as first letter lowercase, then using Capitalised English names. Example : private string customerName;
- Classes which are constructed using the default constructor and which subsequently require lots of calls to setXXX() methods to initialize them should be avoided.
- Member functions should be declared using capitalized (except for the first word) English names.

3.11. Expressions and Statements

For the loop constructs, break and continue should be used rather than return. The use of break is particularly important in the switch statement as code execution will by default continue into the next block.

The starting brace should be at the end of the condition. The ending brace must be on a separate line and aligned with the conditional. All conditional constructs should define a block of code even for single lines of code. The exception to this rule is where the closing and opening brace share the same line as the 'else'.

Place the if / else keyword and conditional expression on the same line. The While construct uses the same layout format as the IF construct. The Switch keyword should appear on its own line, immediately followed by its test expression. The statement block is placed on the next line. Always ensure that a switch statement has a default case.

The try/catch construct is similar to the others. Try keyword should appear on its own line; followed by the open brace on the same line; followed by the statement body; followed by the close brace on its own line. Any number of CATCH phrases are next consisting of the CATCH keyword and the exception expression on its own line; followed by the CATCH body; followed by the close brace on its own line. The FINALLY clause is the same as a CATCH.

3.12. Exception Handling

- Always code for the non happy day scenario.
- Use unchecked exceptions for system exceptions.
- Use checked exceptions for application exceptions. Although, in the welcome package application we have used application exceptions/bussiness exceptions as unchecked exceptions.
- Only catch an exception if you can do something with it or you need to wrap it in something more meaningful to a higher level in the application.
- Log an exception as early as possible.
- Provide useful information when throwing an exception.
- Do not use System.out.XX or System.err.XX unless you have a very good reason.
- Never use the clause 'throws Exception' in a method.

3.13. Memory Management

Java provides it's own memory handling mechanism called the Garbage Collector. Programmers should be aware of how this works. Be aware of

- The implications of String concatenation (and the alternative methods using StringBuffer) and other operations on immutable types.
- Create arrays, buffers etc at their anticipated size to reduce re-sizing overheads.
- Don't create unnecessary objects e.g. In loops don't

3.14. Portability

Java is inherently portable. Programmers should never use operating system specific features.

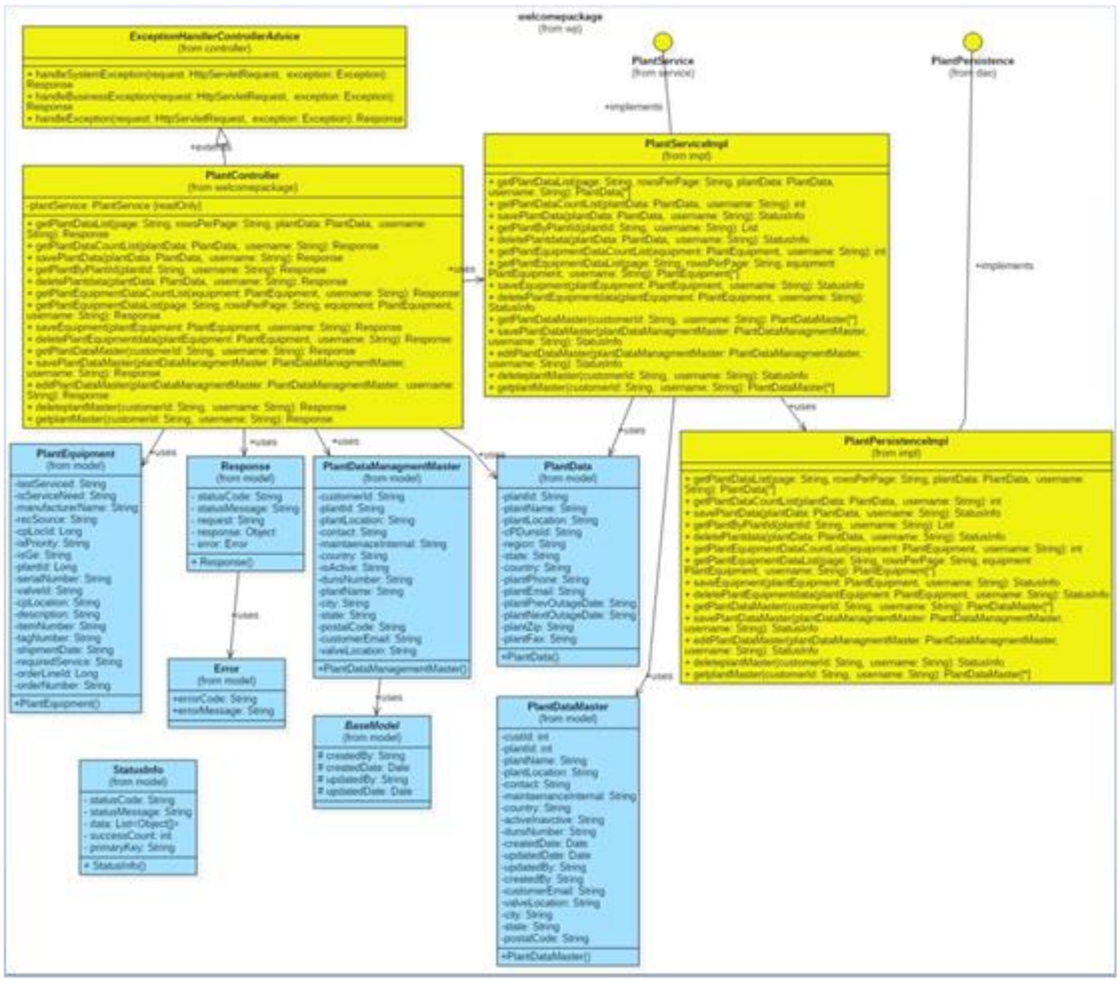
Section 4. Welcome package Standards

4.1. Introduction

This section introduces the changes made as a part of migration activity of welcome package project and the standard followed in the same.

Welcome package project was build on spring MVC framework and contained all the api end points in on single serviceImpl class. As a part of migration all the api end points have been segregated into numerous modules based on their functionality.

4.2. Architecture



The above digram depicts a high level architecture diagram of how welcome package application is going to be implemented on new platform.

The whole application follows a model-service-controller-view architectural flow.

All the requests are handled by corresponding controllers defined within the package *com.ge.fpt.welcomepkg.<<module_Name>>.<<controller>>* .

According to the fuctionality the application contains numerous controller which will handle various bussiness requests. Namely, *PlantManagementController.java* , *UserManagementController.java*, *OrderManagementController.java e*tc.

All the exceptions are handled in a common class named *ExceptionHandlerControllerAdvice.java* defined *package com.ge.fpt.welcomepkg.exceptionhandler.*

According to the controllers we have defined corresponding service/serviceImpl and persistence/persistenceImpl methods to define and carry out bussiness operations required.

A base model class has been implemented which can be extended by other classes for common attributes.

A common Status class named as "*StatusInfo.java*" is defined which conveys information about some particular method to controller layer.

An Error class is defined which will carry information about and error or exceptions occurred within the application. It will be a part of response Object.

4.3. REQUSET/RESPONSE STRUCTURE

The view part of the welcome package is defined by using angularJs technology . All the communication to and from the view and service layer will be done via a stable request-response architechure.

All the requests will traverse via controller layer. All the responses from the service end point will adhere to a common response format defined in the class *ResponseDTO.java.*

```
public class ResponseDTO implements Serializable {
        private String status;
        private String request;
        private Object response;
        private List<Error> error;

        <getters & setters>
        <hashCode() , equals() & toString()>
}
```

- The status object will contain success/failure status of the endpoint request hit.
- The request object will contain the request object which will contain the rquest url,request parameters etc ; of the request being processed.
- The response object will hold the data that the endpoint will return after hitting the service layer.
- The error object will hold error or exceptions, if any.

4.4. UI- LAYER STANDARDS FOLLOWED

Please refer below link for UI coding standards:

https://ge.box.com/s/4vm5gxqqpe1jqur7p1n59ztt0fvj38sn

4.5. CONTROLLER LAYER STANDARDS FOLLOWED

The controller layer is named according to the type of requests it handles .

- All the requests/endpoints which are required for plant management are present in *PlantManagementController.java.*
- ALl the controllers return the ResponseDTO object.
- All the controllers are placed in package *com.ge.fpt.welcomepkg.<<module_name>>.controller* .
- Camel Case nomenclature has been followed for anming the controllers.
- The controllers make use of annotaion based defination to identify and process REST architecture and request-response processes.
- *@RestController* annotaion is used to define controller classes.
- Dependency to-from service classes has been injected by using *@Autowired* annotaion.
- Necessary class level and method level comments has been provided.

4.6. Service layer standards

The service layer has been impelmented by making use of factory menthod design pattern where all the bussiness processes are defined in an interface and the implementing class overrides the required methods.

- The service interface and serviceImpl classes follow camel case nomenclature.
- The service interface and serviceImpl are deifined within base package *com.ge.fpt.welcomepkg.<<module_name>>.service* and package *com.ge.fpt.welcomepkg.<<module_name>>.service.impl* Class level and method level comments are provided.
- Class level Persistence class objects has been defined which is used across the methods to do a persistence class method calls.
- *@Service* annotaion is used to define service classes.
- Dependency to-from persistence classes has been injected by using *@Autowired* annotaion.
- Common logger has been implemented in each method .
- Bussiness excepetions , System exceptions and other errors are handled in try-catch blocks.
- All the exceptions are wrapped up with proper messages and are propagated to the higher layers.

4.7. Persistence layer standards

The persistence layer has been impelmented by making use of factory menthod design pattern where all the bussiness processes are defined in an interface and the implementing class overrides the required methods.

- The persistence interface and persistenceImpl classes follow camel case nomenclature.
- The persistence interface and persistenceImpl are deifined within package *com.ge.fpt.welcomepkg.<<module_name>>.dao* and package *com.ge.fpt.welcomepkg.<<module_name>>.dao.impL* respectively
- Class level and method level comments are provided.
- The persistence classes make use of Jdbc templates to process queries.
- Queries are defined in a file named *sql-query.properties*.
- *@Component* annotaion is used to define persistence classes.
- Common logger has been implemented in each method .

- Bussiness excepetions , System exceptions and other errors are handled in try-catch blocks.
- All the exceptions are wrapped up with proper messages and are propagated to the higher layers.
- The model classes binded with datasource/persistence layer implement *mapRow()* methods to map coloumns/rows to corresponding attributes defined in the entity classes.

4.8. MODEL/ENTITY LAYER STANDARDS FOLLOWED

- The model classes are simple pojo classes which contain crisp data definations.
- Model classes are defined in package *com.ge.fpt.welcomepkg.<<module>>.model* package.
- All the model classes should implement a no-arg constructor along with required parameterized constructors.
- Getters and Setters are define for all the properties/variables defined within the model classes.
- Ovverride hashcode(), equals() & toString() methods.

4.9. DB STANDARDS FOLLOWED

- A spring datasource has been used for database related action and usage.
- *spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.ImprovedNamingStrategy* has been used for naming.
- All the properties and configuration values related to database has been kept in *application.properties* file.
- Necessary comments has been provided in the *application.properties* file.

4.10. CONFIGURATION STANDARDS

- All the required dependencies are added in *pom.xml*.
- A *logback.xml* has been maintained for LOGGER configuration and definations.
- *application.properties* file is maintained to set and define application specific urls, channel_connection app id, proxy port etc.
- *sql_query.properties* has contains all the queries utlized in the application.
- package *com.ge.fpt.welcomepkg. <<module_name>>.service* has been defined to hold the Junit test classes.

- The application makes use of Junit 4 and Jmockit added features to implement unit testing .
- The main method for the application is defined in *WelcomPackageApplication.java* class which is under package *com.ge.fpt.welcomepkg* package.
- Custom exceptions are handled by a *ExceptionHandlerController Advice.java* class which is defined in package *com.ge.fpt.welcomepkg.exceptionhandler* . The makes use of annotaion based configuration to handle exceptions thrown by service api endpoints.
- Example : *@ControllerAdvice*, *@ExceptionHandler* .
- Bussines Exceptions,System Exceptions and other custom exceptions are defined under package *com.ge.fpt.welcomepkg.exception*.
- package *com.ge.fpt.welcomepkg.util* contains application specific utility classes defined for the application like *Messages.java* , *ConnectionFactory.java* etc.

4.11. Logging Standards

- A logback.xml file has been deifined for the application to log various information. Two separate files have been defined to log App based information and framework based informations .
- Namely, *APP_FILE, FRAMEWORK_FILE* and *SERVICE_LOG_FILE*.
- A logs folder has been maintained in */WelcomPkgOperation/logs* which contains all the logs generated by the application.
- A *ch.qos.logback.core.rolling.RollingFileAppender* is used to roll over to log files in both *APP_FILE* and *FRAMEWRK_FILE*.
- Maximum file size defined is *10MB* for each log file.
- *%d{MM-dd-yyyy|HH:mm:ss} [%t] | %-5p | %c | %M | %L | %m%n* pattern used.
- Two properties have been defined in application.properties file to control logging.Namely : *logging.level.org.springframework* and *logging.level.root*.
- Every util class , serviceImpl ,persistenceImpl class and controller class define a logger object and implement logging.
- For logging the entry and exit of methods *_LOGGER.info* level is used .
- For logging the exceptions *_LOGGER.error* level is used.

4.12. Exception Handling

- For Handling exceptions , a base handler has been defined named *ExceptionHandlerControllerAdvice.java*.
- It handles System exceptions , Bussiness exceptions and Exceptions which are thrown by api endpoints by implementing *@ExceptionHandler* annotaion.
- All the exceptions and errors are set in error object defined in *ResponseDTO.java* and are the propagated forward to UI layer.
- System Exceptions, Bussiness Exceptions and WrappedRunTimeExceptions are defined in package *com.ge.fpt.welcomepkg.exception*.

- Methods propagate excepetions by using try-catch blockes defined for each type of exception that may occur.
- Loggers are defined within each catch block to log the exception message and type of exception with trace.
- *systemException.setExceptionlogged()* has been set to true wherever exception has been logged.
- Execptions have been classified and codes have been assigned . The list of codes is maitained in *exception-code.properties* and *exception-code-detail.properties.*

Section 5. Appendix