Extensive Code Review Checklist

Functional Review

The implemented functionality should comply with the criteria outlined in this section.

Functionality works as described

First check whether the code performs the way it's described in the related issue.

- Read and understand the issue description.
- Understand the purpose of the issue: why it was created, what's the goal behind it, what is the expected outcome.
- Verify the implementation against the description: the bug should be fixed and no longer reproducible.
- Verify that the issue goal is met and the expected outcome is achieved.
- Consider if the described functionality can be improved. If so, add comments to the JIRA's respective pull request (PR) or suggest improvements.

Related functionality is identified and tested

Make sure the new changes don't impact the way application works. To identify the areas possibly affected:

- See whether the code references the changed functionality.
- Check bundle dependencies and whether they are in sync (e.g. if WorkflowBundle has been changed, check all areas where changed classes/services from this bundle were customized or used).
- Make a visual check (e.g. if a UI element was changed, similar elements should be changed respectively).

Manual check throughout the production environment

Manually verify that the functionality works properly across the production environment and that all the issue requirements are met. The production environment may include:

- Setting up Cron;
- Running at least one message queue consumer;
- Running optional and integration services;

Architectural Review

Review how the architecture of the implemented solution interrelates to the overall application architecture.

Follow SOLID, KISS, DRY, GRASP and YAGNI Principles

The solution should follow these principles. If not, add a comment using suggested wording.

SOLID

- **Single responsibility principle**. A class should have only one responsibility (i.e., only one potential change in the software's specification can affect the specification of the class).
- **Open/closed principle**. Software entities are open for extension but closed for modification.
- **Liskov substitution principle**. Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Interface segregation principle**. Many client-specific interfaces are better than one general-purpose interface.
- **Dependency inversion principle**. Rely upon abstractions, not concretions.

Sample comment referencing principles, "Looks like this class has two responsibilities; it parses files with configuration and performs actual requests to DB. Please split this class into two separate classes according to single responsibility principle. See https://en.wikipedia.org/wiki/Single_responsibility_principle."

KISS

The 'Keep It Simple Stupid' principle implies the solution should be simple rather than complex. *A comment may read, "Please replace these embedded arrays with the one level array (KISS). It'll be easier to understand for other developers and the community."*

DRY

The 'Don't Repeat Yourself' principle reduces repetition of any kind of information. Avoid copy-pasting solution and don't introduce a new approach in addition to the existing one. *"Looks like the same code for statistics calculation is already implemented in some.service.name. Please avoid copy-pasting (DRY) and reuse this service here."*

GRASP

The 'General Responsibility Assignment Software Patterns (or principles)' are guidelines for assigning responsibility to classes and objects in object-oriented design. Patterns and principles used in GRASP are:

- Controller
- Creator

- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

All of these patterns address a specific software problem; each of them common for almost every software development project. These techniques better document and standardize old, tried-and-tested programming principles in an object-oriented design. *"After the changes made in this task, this class seems to highly couple with WorkflowBundle. Would be nice to minimize interaction with WorkflowBundle and extract it to a separate class to maintain loose coupling between this functionality and the workflows."*
YAGNI
The "You Aren't Gonna Need It" principle reminds you to implement things only when you **need them**, not when you think you **might need** it later. Determine if any part of the solution can be removed or simplified without failing to satisfy the requirements. *"This class is not used anywhere except the tests and CLI command to be removed, too. Please remove this class, all related tests and CLI command to reduce the amount of supported code (YAGNI)."*

Functionality is implemented using relevant architectural design

Verify that the implementation uses relevant architectural design (MVC, bundle, component). If you are unfamiliar with the relevant design, consult the tech lead or architect. If you're reviewing a bug fix, make sure the implementation provides fixes for the source of the issue, not its consequence.

No duplicate implementations

Verify that the approaches applied in the PR don't offer new resolutions to the existing issue. Examples of redundant/duplicate code include:

- New interfaces or abstract classes that are not related to the new functionality
- Code in the package that is out of scope of the current JIRA ticket
- Copy-pasted code/different coding styles delivered by the same person
- Class name doesn't reflect its responsibility

Extensibility points

The majority of delivered components should be extendable out of the box. Make sure that the solution maintains a reasonable extensibility. The extension points you may use include:

Extension via application configuration

- Global level configuration - usually files at app/config
- Bundle level configuration - usually files at <BundleName>/Resources/config/oro

Extension via DI container

- Parameters overriding
- Service decoration
- Service overriding
- Calls on the services (e.g., to inject other service)
- Tags
- Compiler passes

Extension via implementation

- Open/closed principle of SOLID
- Dependency injection
- Interfaces and abstract classes
- Events

The best way to identify whether the implemented solution has enough extension points is to consider how you could possibly customize it. If nothing crosses your mind, turn to your fellow developers.

Implementation Review

Below are the evaluation criteria for the task implementation.

Code defects

There must be no logical errors in the code. Apart from its intended purpose, the code should prevent possible misuses and logical issues. For example, consider the following condition:

```
if ($this->isActive() && $this->calculateStatus() !== 'inactive') {
    // ...
}
```

This condition has a hidden issue. Method *calculateStatus* may change the state of a variable (in memory, in DB or in cache) but this code only execute if the first part of the condition is true. This condition might lead to inconsistent and unstable behavior if other application areas rely upon its functioning. One possible solution would be to implement nesting conditions.

The external code (i.e., framework and libraries) should be used as intended. For example, a coder may use *Symfony\Component\HttpKernel\Exception\NotFoundHttpException* exception inside a model layer code. However, this is strictly HTTP exception to be only used at the HTTP request processing (e.g., in controller). A possible solution would replace this exception with a standard *LogicException or RuntimeException* or create a custom exception, catch it and cast as *Symfony\Component\HttpKernel\Exception\NotFoundHttpException*.
If the code contains exceptions, verify that they belong to the proper namespace. The exception message should provide enough information for a developer to understand exactly what went wrong. Feel free to add additional parameters if needed (e.g., variable type, template name).

The code and all of its elements such as class names, variables or comments must be free of grammatical errors and typos. If you are unsure of grammar, check your code grammar using external sources, if this does not violate any Non Disclosure Agreement in place.

Security vulnerabilities

Review the implementation source code for security vulnerabilities and ensure it's protected against security attacks such as:

- SQL Injection where malicious SQL code is passed as a part of original query. To avoid this issue, always pass arguments as query parameters or ensure they are manually validated and/or escaped.
- Cross-Site Scripting (XSS) where malicious code in the form of a client side script is sent by an attacker via the web application. To ensure the required protection, properly filter all input data and escape all rendered data.
- Cross-Site Request Forgery (CSRF) attacks force a user to execute unwanted actions on a web application in which they're currently authenticated. Use CSRF tokens as a basic prevention method.

This list of the most common application security attacks may help you when determining whether the code is vulnerable to malicious actions.

Also carefully review the following code fragments:

DBAL/Doctrine requests Make sure all parameters aren't passed directly to query or are properly escaped
Value rendering in templates. Values should be escaped using escape filter or autoescape tag.

Forms always contain CSRF token. Check this manually in the HTML code for the hidden field csrf_token.
Any application part that uses ACL (pages, grids). Ensure that a user can only access allowed resources:

- Check that ACL is applied (e.g., manually disable it).
- Check that proper business unit and organization restrictions are applied (e.g. if the Organization ACL level is used, the user should be able to access only resources from their organization).

In addition, these are the most common PHP level vulnerabilities to protect against:
Eval injection. Where the usage of eval function in the original code is prohibited.
Usage of superglobal variables. Don't use *$_SESSION* and other superglobal variables directly.

Use security:check command if you include 3rd party libraries to the composer. This will ensure they don't introduce known vulnerabilities.

Also consider potential security breaches, e.g., data-sensitive information in error messages, suspicious user activities, etc. Ensure that all passwords are salt hashed with the PHP password hashing API. Each application has a unique hash stored in *%secret%* parameter that can be used to provide extra protection in case of data leak.

Memory consumption

The implementation should optimize memory usage and prevent memory leaks. Here are examples of common use cases where the issue of high memory consumption or memory leaks may occur:

Processing of big arrays. You may work directly with the entities from Unit of Work as the code developer merges all modified arrays. The new merged array will consume extra memory and this may impact performance. To resolve the issue, process changes separately or use generators.

Doctrine hydration. Doctrine entities come in handy during regular usage; however, hydration processes are time and memory consuming. It's not a problem when extracting several entities but is not suitable for a greater number of entities, especially if the query contains additional *JOIN*s to entities to be hydrated. To resolve the issue, try the following:

- Hydrate data to array (i.e., using the method getArrayResult).
- Select partial objects.
- Implement custom hydrator (only if objects are required).

Request of storage data. A large amount of data being fetched at once (e.g., when the developer extracts all entities from DB) may be memory consuming. This affects all storage types. Possible solutions to consider could be:

- Batch iterating (e.g., using *BufferedQueryResultIterator*).
- Iterate manually row-by-row (using the DB cursor).

Memory leak might appear in any code that processes a large amount of data. If you work with these functionalities, try not to increase memory consumption. Doctrine uses a lot of memory, so if you need to handle a great amount of its entities be sure to clear required EntityManager(s).

Performance issues (code and storage)

Performance issues occur when changes to the code under review make it impossible to meet certain requirements of the product owner (e.g., time or memory limit). The product owner should specify performance requirements in JIRA. Performance issues can be the following types:

Code performance issue. The code increases execution time or is memory consuming. For example, the new implementation performs full hydration of entities using Doctrine, which increases request time from 0.5 to 0.8 seconds. Verify that the entities are really required in this case and try to minimize the impact of the hydration (don't hydrate all related entities or select partial objects). Consider applying KISS and YAGNI principles first.

Storage performance issue. Poorly designed queries to DB or other storage.
For example, with usage of subquery with *IN* condition i.e., *WHERE IN (SELECT …)* - some RDBMSs execute this subquery for each row in a result set. Pay attention to all complex queries generated by Doctrine as it may build inefficient and slow queries.

You can optimize the queries by:

- Decreasing the amount of selected data in one query (e.g., use batch iteration, but compare if batch processing is faster than one query).
- Rebuilding the entire query (e.g., replace *WHERE IN* (*<subquery>*) with *LEFT JOIN* (*<subquery>*) This fix might not work if Doctrine ORM is used.
- Adding DB level index(es) that will speed up search (*WHERE* conditions) but will slow down changes (*INSERT*, *UPDATE*).
- Replacing one long query with several shorter queries.
- Denormalizing storage structure (if required, copy some values from one table to another to avoid extra *JOINs*.

You may also face other performance issues not related to code or storage such as external API restrictions. First, detect bottlenecks and then consider possible solutions.

Logger

The application must log all crucial actions and exceptional situations. This includes:

- Business operations with detailed context for every step.
- Transitions: workflow transition, user banned, checkout completed.
- Integration points: calls, availability, response time.
- Resources availability: limit reached, capacity exhausted.
- Service availability: startup, shutdown, response time.
- Input and output (if it helps to find the issue).

Exceptional situations should also be logged (e.g., catch of an exception or manual conditional check). This information is important because it enables quick identification of the source of the problem.

Boundary values

Always check boundary values for conditions containing specific boundary values (*$value >= 100*). Check the exact value, the value that is slightly lower and value that is slightly higher than the exact value (e.g., *99, 100 and 101*). This will ensure that condition is fully covered by the test. You can also check if the unit test for the required class contains all boundary value checks.

Code readability

As a rule of thumb, if it takes you longer than 5 seconds to understand the code, readability must be improved. Here are ways to make code more readable:

- Use self-explanatory names for classes, methods and variables.
- Avoid comments if you can use a proper naming for explanations.
- Comment code that can't be self-explanatory (for example, add a simple comment before the complex regexp explaining what it does).
- Use consistent padding to align array values, regular variables and comments.
- Group your code in line with the functionality. Grouping can be moved to separate methods and improve readability.
- Avoid long methods. If the method has over 80 code lines, split it into separate methods.

- Avoid deep nesting. If the code has more than 3 nesting levels, move some of them to separate methods).
- Avoid small abstractions.

Private services in DI container

Check that added or modified services were marked as private. It's not possible to extract private services directly from DI container, they can only be injected into other services as dependencies. Private services consume less memory during the initialization and decrease the overall container size. Private services partially maintain open/closed principle because they can be only used only internally (like private properties or methods).

Some services (e.g., form types and event listeners) can't be marked as private because DI container is used during their initialization. To check whether a service is private, consider calling from controller or CLI command. Should this be impossible, this service will is only required as a dependency of other services and can be defined as private.

Translatable static data

Make sure that page static data, any string that can't be changed by users from the UI, is translatable. To check that all static data is properly translated, use debug translator. It will wrap all translatable strings and keep untranslatable strings intact.

Docblocks for classes and methods

Make sure that docblocks for class and methods were properly defined by the code developer. Docblock for class (or interface) is obligatory. It should describe the main class purpose, responsibility and the most common use cases. If you're not sure how to describe a class, you can remove it.

Docblock for a method is obligatory if the method has arguments or returns a value. It must describe types of arguments/result and their structure (e.g., if method accepts an array or returns an array in a specific format) and provide an additional description, if necessary. Docblocks may also use {@inheritdoc} tag not to copy/paste all data from the parent method or interface.

```
/**
 * Interface for a result record (row) from a datagrid, allows to access record data
 */
interface ResultRecordInterface
{
    /**
     * Get value of record property by name
     *
     * @param  string $name
     * @return mixed
     * @throws LogicException When cannot get value
     */
    public function getValue($name);

    /**
     * Get root entity of current result record
     *
     * @return object|null
     */
    public function getRootEntity();
}
```

Automated Tests

Below are the recommendations for automated tests review.

Unit tests: class level

Unit tests should cover all PHP code instances, except for controllers and classes responsible for interaction with storage. Unit tests should test only class level functionality. They must not check interaction with other classes (functional tests will do that). If it's hard to unit test some specific classes, these can be covered with a functional test. Unit tests should:

- Ensure 100% test code coverage.
- Cover boundary values.
- Have self-explanatory names.
- Check one action each.
- Have no interaction with storage (file system, DB).
- Be environment-independent.

Functional tests: application level

Functional tests check the entire functionality with the pre-set application. They depend on the environment (storage, DI container, etc.). Functional tests are implemented for code that can't be covered with unit tests (such as controllers and entity repositories) and check the processing cycle from request building to request verification. They can be also used instead of unit tests to test class or service. In this case, the functional test should match all criteria from unit tests such as 100% coverage and boundary values check.

Functional tests don't test Javascript logic but rather process the response data.

They are executed in a test environment that must mimic the production mode. While the test environment might differ slightly from the production environment, the tests must reflect real cases a user may face in the production environment.

Functional tests may test full scenarios/workflows and use annotation *@depends* to arrange a chain of tests into a scenario. However, if steps of this workflow can be used separately, they must be checked separately (e.g., CRUD operations).

Functional tests coverage

Functional test should be based on the real use cases. They should also cover negative use cases and check possible errors (e.g. from validations), exceptions (during the code processing) and general application stability (the application still returns a user-friendly response even when an error has occurred).

Functionality fully covered with tests

Verify that all implemented backend functionality is fully covered by either unit or functional tests. If unit test covers all possible user inputs, functional test can check for only one or two of the most common cases. The following use cases can't be fully covered by unit or functional tests:

- Javascript logic at frontend (can be covered by behat tests instead).
- Multi-process logic (e.g., parallel consumers, race conditions or deadlocks).
- High-load use cases (performance checks).

As you're working on a story, task, improvement or bug, always verify that your changes are covered with tests.

Documentation

This section describes all types of documentation required to complete a task.

Developer documentation

Make sure that the new functionality is properly described in bundle documentation. Technical details of the implementation to be documented include:

- **Feature description**. Provide an overall description with screenshots illustrating what the feature looks like on the front end.

- **Feature usage examples**. Describe how feature is to be used in production with real use cases.
- **Feature configuration**. Describe the structure and options (with default values) and give examples of the most common use cases with detailed value explanations.
- **CLI commands**. Describe command responsibility, its arguments/options, option purposes and examples of how to run the command with the actual output.
- **API resources**. Describe available resources and give the most common use examples.
- **General purpose interfaces**. Describe the responsibility of interfaces and their methods. Provide examples of implementations with real use cases.
- **Algorithms.** Describe unusual or complex algorithms that were used and include the link to the complete specification.
- **Data structures.** Document unusual or complex data structures that were used with a link to full specification (if possible).
- **Useful links**. Include a list of references to used libraries and components, manuals and other related information.
- **Other technical details.** Don't forget to include information such as environment setup, recommendations on testing and debugging.

Documentation will be authored by community developers with a different level of expertise, so it makes sense to explain even common things or at least add links to external resources that provide the detailed description.

Solution extendability

Each feature must be extended when required. Other developers should be able to inject their own logic or override the existing one. Verify that all introduced ways to extend the functionality are documented and provide at least one example each. You can also check whether common ways are described (e.g., service decoration or parameter overriding), especially if these might be required in production.

User documentation code examples

Code examples are given not only in the developer documentation, but also in the user documentation (OroCRM and OroCommerce). It the monolithic repository, user documentation is stored in the documentation directory.

Make sure the code examples in user documentation are up-to-date and contain enough code to explain the described behavior to users. The related description may also require additional fixes.