****

**General Coding standard**:

**1**. Use CSS and Javascript versioning in each deployment, if there is any changes.  
 e.g:

<script src="<?php echo $mosConfig\_live\_site;?>/components/com\_financialnew/js/application\_validate.js?v=<?php echo $mosConfig\_css\_version; ?>"></script>  
Note: $mosConfig\_css\_version will contain current date , if we will change anything in code.

**2**. Don’t write Inline CSS and JavaScript. Please write css and js code in separate .js/.css file.

**3**. Don’t use below

ini\_set("max\_execution\_time", 0);

ini\_set('memory\_limit','2048M');

if necessary, please take approval and discuss advantage and disadvantage.

**4**. Don’t store huge data in SESSION , instead use generic non-prod/prod Redis cache.

**Non-prod redis cache URL**: b2c-lambda-redis.usz10j.ng.0001.use1.cache.amazonaws.com:6379  
 **Prod redis cache URL**: b2c-lambda-cache-services-prod-redis.usz10j.ng.0001.use1.cache.amazonaws.com:6379

**5**. Use typecast always before using user input or post input like:

$userid = (int) $post['customer\_id'];

Also check

$userid > 0, $appid > 0 and $matchid > 0

**6**. Always check value is empty or not and sanitize that value and then pass to SQL query(escape).

**7**. Don’t write or add new function in web.function.php under replica, because its already huge file

**8**. Don’t write separate SQL query for fetch single column value from table , use inbuilt method

**getTableValue**(); which is very efficient.

**9**. Don’t use PHP Mail or SMTP mail , instead use Sendgrid middleware mail functionality.

**10**.Dont use inline config variable/array/url inline, use configuration /env file for that, it will always help to change value without whole deployment

**11**. Maintain history in log file (S3)/DB of each edit of any page, if possible. Also log each third party API on each step with unique request id.

**12**. Please also verify **console log error**, if it impacts page performance.

**13**. Please use CDN URL of any images instead of static images folder.

**14**. SQS Guideline

**SQS Type**:

A. Standard SQS

B. FIFO SQS

**Difference b/w Standard and FIFO**:

**Message Order**

Standard queues provide best-effort ordering which ensures that messages are generally delivered in the same order as they are sent. Occasionally (because of the highly distributed architecture that allows high throughput), more than one copy of a message might be delivered out of order.

FIFO queues offer first-in-first-out delivery and exactly once processing: the order in which messages are sent and received is strictly preserved.

**Delivery**

Standard queues guarantee that a message is delivered at least once, and duplicates can be introduced into the queue.

FIFO queues ensure a message is delivered exactly once and remains available until a consumer processes and deletes it; duplicates are not introduced into the queue.

Standard queues allow nearly unlimited number of transactions per second.

FIFO queues allow to process up to 3000 messages per second per API action.

**Regions**

Standard queues are available in all the regions.

FIFO queues are currently available in limited regions only.

Standard Queues are supported by all AWS services.

FIFO Queues are currently not supported by all AWS services like: CloudWatch Events, S3 Event Notifications, SNS Topic Subscriptions, Auto Scaling Lifecycle Hooks, AWS IoT Rule Actions, AWS Lambda Dead Letter Queues.

**When should use FIFO or Standard**

Choose Standard queues if:

Standard queues can be used in any scenarios, as long as the application can process messages that arrive more than once and out of order.

Decouple live user requests from intensive background work: Let users upload media while resizing or encoding it.

Allocate tasks to multiple worker nodes: Process a high number of credit card validation requests.

Batch messages for future processing: Schedule multiple entries to be added to a database.

**Choose FIFO queues if**:

FIFO queues are designed to enhance messaging between applications when the order of operations and events is critical, or where duplicates can’t be tolerated.

Ensure that user-entered commands are executed in the right order.

Display the correct product price by sending price modifications in the right order.

Prevent a student from enrolling in a course before registering for an account.

**Setting of Each SQS**

**Visibility timeout** – The length of time that a message received from a queue (by one consumer) won't be visible to the other message consumers

The default visibility timeout for a message is 30 seconds. The minimum is 0 seconds. The maximum is 12 hours.

**Message retention period** – The amount of time that Amazon SQS retains messages that remain in the queue. By default, the queue retains messages for four days. You can configure a queue to retain messages for up to 14 days.

**Delivery delay** – The amount of time that Amazon SQS will delay before delivering a message that is added to the queue.

Use case of Delay: Delay queues let you postpone the delivery of new messages to consumers for a number of seconds, for example, when your consumer application needs additional time to process messages. If you create a delay queue, any messages that you send to the queue remain invisible to consumers for the duration of the delay period. The default (minimum) delay for a queue is 0 seconds. The maximum is 15 minutes

**Delayinseconds**: To set delay seconds on individual messages will only work on Standard SQS not on FIFO.

Enable content-based deduplication – Amazon SQS can automatically create deduplication IDs based on the body of the message, there is no need for Standard SQS, but for FIFO , we have to sent a unique id for this attribute.

Proper log in CloudWatch or S3 bucket with request and response

B2C Replica already have generic method of Standard SQS (FIFO will not support our current PHP version 5.3) under includes/web.fnction.php:

**function initiateSQS($params = array()){**

// Params are

1. AWS Region

2. QueueUrl

3. Message (JSON stringigy)

// Response

It will return a message id as string. If possible please save this message id for that request.

}

**Size limit: 256 KB (JSON payload)**

**15. SNS Guideline**

Amazon SNS is a fast, flexible, fully managed push notification service that lets you send individual messages or to bulk messages to large numbers of recipients. Amazon SNS makes it simple and cost effective to send push notifications to mobile device users, email recipients or even send messages to other distributed services.

SNS is a distributed publish-subscribe system. Messages are pushed to subscribers as and when they are sent by publishers to SNS.

**KEY difference b/w SQS and SNS**

**Message consumption**

SQS : Pull Mechanism — Consumers poll messages from SQS.

SNS : Push Mechanism — SNS pushes messages to consumers.

**Persistence**

SQS : Messages are persisted for some duration is no consumer available. The retention period value is from 1 minute to 14 days. The default is 4 days.

SNS : No persistence. Whichever consumer is present at the time of message arrival, get the message and the message is deleted. If no consumers available, then the message is lost.

In SQS the message delivery is guaranteed but in SNS it is not.

**Consumer Type**

SQS : All the consumers are supposed to be identical and hence process the messages in exact same way.

SNS : All the consumers are (supposed to be) processing the messages in different ways.

**Use Cases**

**Choose SNS if**:

You would like to be able to publish and consume batches of messages.

You would like to allow same message to be processed in multiple ways.

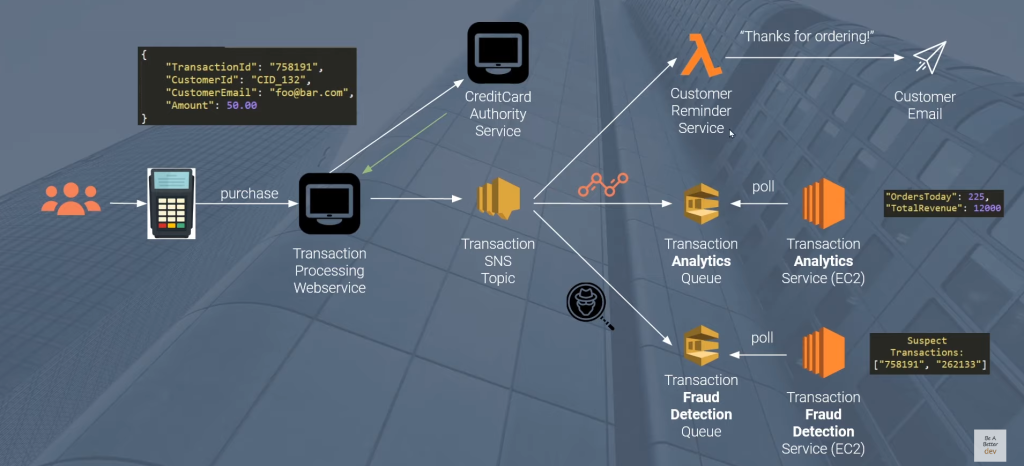
Multiple subscribers are needed.

**Choose SQS if**:

You need a simple queue with no particular additional requirements.

Decoupling two applications and allowing parallel asynchronous processing.

Only one subscriber is needed.



Already common function built in B2C Replica under includes/web.function.php

**function initiateSNS($queue\_name, $payloadData, $appid){**

// params

a. Topic ARN (AWS Resource Name)

b. Message (JSON Payload)

c. AWS Region

//Response

It will return a message id, we should store it for futher debugging purpose.

}

**16**. Use AWS S3 for Log management

**17. Lambda Guideline**

AWS Lambda is an event-driven serverless computing platform.

This means that it runs code in response to events (“event-driven”), while automatically taking care of all the back-end infrastructure and admin that is needed to run your code (“serverless”).

How Does AWS Lambda Work?

When you send your code to Lambda, you are actually deploying it in a container. The container, however, is itself created, deployed, and managed entirely by AWS. You do not need to be involved in the creation or management of the container, and in fact, you do not have access to any of the infrastructure resources that might allow you to take control of it. The container is simply a location for your code

**What Makes AWS Lambda Different?**

**a. Pay-as-you-go for great cost savings**

Just like other public cloud services, Lambda has a pay-as-you-go pricing model with a generous free tier, and it is one of the most appealing for cost savings. Lambda billing is based on used memory, the number of requests and execution duration rounded up to the nearest 100 milliseconds. This is a huge leap for fine-grained billing in order not to pay for spare compute resources compared to the second based billing of EC2

**b. Completely event-driven**

While most of the PaaS offerings are designed to be running 24/7, Lambda is completely event-driven; it will only run when invoked.

This is perfect for application services which have quiet periods followed by peaks in traffic.

**c. Fully scalable**

When it comes to scalability, Lambda can instantly scale up to a large number of parallel executions, controlled by the number of concurrent executions requested. Scaling down is handled by automatically; when the Lambda function execution finishes, all the resources associated with it are automatically destroyed.

**Limitations of AWS Lambda**:

However, AWS Lambda is not a silver bullet for every use case. For example, it should not be used for anything that you need to control or manage at the infrastructure level, nor should it be used for a large monolithic application or suite of applications.

Lambda comes with a number of “limitations”, which is good to keep in mind when architecting a solution.

There are some “hard limitations” for the runtime environment: the disk space is limited to 500MB, memory can vary from 128MB to 3GB and the execution timeout for a function is 15 minutes. Package constraints like the size of deployment package (250MB) and the number of file descriptors (1024) are also defined as hard limits.

Similarly, there are “limitations” for the requests served by Lambda: request and response body synchronous event payload can be a maximum of 6 MB while an asynchronous invocation payload can be up to 256KB.

**Are These True Limitations?**

You may be wondering why I keep quoting the word “limitation” - because these should not be viewed as limitations, but as well defined architectural principles for the Lambda service:

If your Lambda function is running for hours, it should be moved to Elastic Beanstalk or EC2 rather than Lambda.

If the deployment package jar is greater than 50 MB in size, it should be broken down to multiple packages and functions.

If the requests payloads exceed the limits, you should break them up into multiple request endpoints.

It all comes down to preventing deploying monolithic applications as Lambda functions, and designing stateless microservices as a collection of functions instead. Having this mindset, the “limitations” make complete sense.

**Setting Parameters**:

When a function returns an error after execution, Lambda attempts to run it two more times by default. With Maximum Retry Attempts, you can customize the maximum number of retries from 0 to 2.

**So retry should be 0 in some case**.

**18**. Serverless Guideline(Please follow):  
 <https://www.serverless.com/framework/docs/providers/aws/guide/serverless.yml>

**19. MYSQL Guideline**

**a**. Use SELECT \* only if needed

**b**. Use ORDER BY Clause only if needed

If you want to show the result in front-end application, let it ORDER the result set. Doing this in SQL may slow down the response time in the multi-user environment.

**c**. Use CHAR (1) over VARCHAR(1)

If you string a single character, use CHAR(1) instead of VARCHAR(1) because VARCHAR(1) will take extra byte to store information.

**d**. Protect Your Script From SQL Injection

If you don't escape your characters used in SQL strings, your code is vulnerable to SQL injections. You can avoid this either by using the mysqli\_real\_escape\_string() or by using prepared statements.

Here's an example of mysqli\_real\_escape\_string() in action:

$username = mysqli\_real\_escape\_string( $GET['username'] );

and a prepared statement:

$id = $\_GET['id'];

$statement = $connection->prepare( "SELECT \* FROM tbl\_members WHERE id = ?" );

$statement->bind\_param( "i", $id );

$statement->execute();

By using prepared statements, we never embed the user's inputted data directly into our query. Instead, we use the bind\_param method to bind the values (and escaping) to the query. It's much safer and, notably, faster when executing multiple CRUD statements at once

**20**. **MongoDB Guideline**

**a**. **Understand query patterns and profiling**

The first step in optimizing performance is to understand expected and actual query patterns.

Once you have a firm grasp of your application's query habits, you can develop your data model and choose suitable indices.

MongoDB provides the explain method to show how MongoDB will execute a given query.

Use MongoDB's profiling capabilities to understand what your application is doing with MongoDb and whether it meets expectations.

**b**. **Review data modeling and indexing**

Develop your application's data model based on MongoDB's capabilities. To do this, you must always understand your schema before starting a project.

If you are unaware of the schema during the development phase, you will often have to redesign things down the road, which can be very costly.

MongoDB is flexible with schemas which is one of the core advantages of the platform.

However, this does not mean you can overlook the data modeling and indexing portion of your project.

You will still need to create indexes that support queries used by the application and monitor accessibility over time.

**c. Ensure you are embedding and referencing**

Embed related objects in MongoDB documents when possible.

This will avoid the performance overhead of repeated requests for data stored in separate collections, which can be much slower than embedded fields.

Through embedding, you will prevent application joins which will decrease queries and updates, thus increasing performance.

MongoDB's $lookup command looks up an object referenced by another document and embeds it directly into this document.

Still, it doesn't reference expansion (for example, looking up a collection with hundreds or thousands of records).

A good practice when using MongoDump scripts to export your MongoDB content is to use findAndModify instead of findOneAndUpdate because these two commands are not equivalent.

The first one returns the modified document after changes have been made.

At the same time, the second does not return any result to the application server, even if some result is returned to MongoDB.

**d**. **Size the memory**

You must be aware of the amount of memory and the MongoDB configuration settings.

When the working set of an application fits in RAM, read activity should be minimal.

However, read activity will begin to rise if your working set outstrips the instance size or server's available RAM.

If you detect this is happening, you may fix it by moving to a more notable instance with more memory.

To manage your MongoDB object cache and avoid out-of-memory errors caused by large or complex queries, you can set size limits on each collection with DB.collection. cacheLimit.

It's important to note that this setting doesn't apply locks, so it won't affect read operations for a single update thread (update operation affects only one document), leading to data inconsistency if concurrent updates are executed against the same documents.

**e**. **Use replication and sharding**

Replication and sharding are some of the core advantages when scaling horizontally using MongoDB and are essential when dealing with high volumes of data for performance.

When using MongoDump scripts to export data from MongoDB servers or at any point in time where massive amounts of documents (millions) are exported into a single file, this can generate an I/O bottleneck because each document must have its own JSON object before being written to the output stream.

Developers can use replica sets to replicate data from a primary server or node across several secondaries.

What this accomplishes is to reduce contention and improve load balancing.

Replication allows MongoDB servers to communicate with each other to remain synchronized even if one of them goes down unexpectedly.

Replica sets provide automatic failover by electing a primary replica set member, which handles all operations sent by clients until it fails.

If this happens, an election process begins where another server will be elected as the new primary based on priority settings defined in the replica set configuration file (MongoDB-rs.conf).

Similar to replication, sharding is another way to distribute large sets of data while improving performance.

Sharding is one of the core advantages that accompanies any NoSQL platform, and MongoDB is no exception.

MongoDB sharding is agnostic to your data nodes and can be replicated as many times as necessary for high availability, making it an ideal solution when you need scalability.

Shards distribute data across different database instances according to specific criteria, which lets MongoDB scale horizontally across servers.

When looking to increase storage and optimize performance, shards are valuable for MongoDB best practices.

**Embedding vs. Referencing**

MongoDB schema design actually comes down to only two choices for every piece of data. You can either embed that data directly or reference another piece of data using the $lookup operator (similar to a JOIN). Let's look at the pros and cons of using each option in your schema.

**Embedding**

**Advantages:**

You can retrieve all relevant information in a single query.

Avoid implementing joins in application code or using $lookup.

Update related information as a single atomic operation. By default, all CRUD operations on a single document are ACID compliant.

However, if you need a transaction across multiple operations, you can use the transaction operator.

Though transactions are available starting 4.0, however, I should add that it's an anti-pattern to be overly reliant on using them in your application.

Limitations

Large documents mean more overhead if most fields are not relevant. You can increase query performance by limiting the size of the documents that you are sending over the wire for each query.

There is a 16-MB document size limit in MongoDB. If you are embedding too much data inside a single document, you could potentially hit this limit.

**Referencing:**

So the other option for designing our schema is referencing another document using a document's unique object ID and connecting them together using the $lookup operator. Referencing works similarly as the JOIN operator in an SQL query. It allows us to split up data to make more efficient and scalable queries, yet maintain relationships between data.

**Advantages:**

By splitting up data, you will have smaller documents.

Less likely to reach 16-MB-per-document limit.

Infrequently accessed information not needed on every query.

Reduce the amount of duplication of data. However, it's important to note that data duplication should not be avoided if it results in a better schema.

Limitations

In order to retrieve all the data in the referenced documents, a minimum of two queries or $lookup required to retrieve all the information.

**21**. **Use DRY Approach (Don't Repeat Yourself)**

**This code...**

$mysqli = mysqli\_connect('localhost', 'reinhold', 'secret\_hash');

mysqli\_select\_db('wordpress') or die("cannot select DB");

**now with the DRY approach**:

$db\_host = 'localhost';

$db\_user = 'reinhold';

$db\_password = 'secret\_hash';

$db\_database = 'wordpress';

$mysqli = mysqli\_connect($db\_host, $db\_user, $db\_password);

mysqli\_select\_db($db\_database);

**21**. Use Meaningful, Consistent Naming Conventions

**22**. **Don't Copy Extra Variables**

For example, the following code snippet:

$description = strip\_tags($\_POST['description']);

echo $description;

can be written into a single line like this:

echo strip\_tags($\_POST['description']);

**23.** Always use US Date format (mm-dd-yyyy) on UI display , if not specified otherwise.

**24**. Please perform unit test at development level.

**25**. Clear names for variables: Always try to put names expressing what you want to do with them

**a**. Keep functions short: Try to write as few lines as possible when writing functions. This doesn’t mean you should be doing functional programming, but try to put small pieces of your code that do a specific task, like getting records from a database, in a separate function.

**b**. Use the keywords TODO, FIXME, NOTE for your comments. Using them forces you to write comments only when something that needs to be completed or fixed is in your code.

**c**. Use comments at the beginning of the file: Describe what the file does, you don’t need to comment each function unless you’re writing an open API, but for internal classes try to keep descriptions only in the top of your document. The functions’ name should express what it does. If you have functions with names GetProductsByEnabledUsers() in your classes, you may need to refactor that into other classes.

**26**. **USE php -l {{filename}} // to check php syntax error**

**27**. Code MUST use 4 spaces for indenting, not tabs.

**a**. There MUST NOT be a hard limit on line length; the soft limit MUST be 120 characters; lines SHOULD be 80 characters or less.

**b**. There MUST be one blank line after the namespace declaration, and there MUST be one blank line after the block of use declarations.

**c**. Opening braces for classes MUST go on the next line, and closing braces MUST go on the next line after the body.

**d**. Opening braces for methods MUST go on the next line, and closing braces MUST go on the next line after the body.

28. **GIT Guidelines**:

**A**. always cut the branch for the task from master/prod (whichever one is production branch)

**B**. always prefix JIRA ID in branch name

**C**. always include date in patch filename

**D**. always mention branch-name in JIRA

**E**. While committing changes in your branch, always check for syntax error by using below command -

**a**) php -l filename.php

**b)** node --check filename.js

**c**) python -m py\_compile filename.py

**F**. while merging on dev/stage/any branch, always use below steps (e.g. on dev) -

**a**) git checkout BLP-123

**b**) git pull origin master

**c**) git branch -D dev

**d**) git remote update

**e**) git checkout dev

**f**) git merge BLP-123

**g**) resolve conflicts if any (also coordinate with the team whose file gets conflict)

**h**) git diff origin/dev (save/review these changes)

**i**) git push origin dev

**G.** Patch file command -  
git diff origin/master > /PATH/BRANCHNAME\_DATE.patch  
git diff origin/master --ignore-space-change > /PATH/BRANCHNAME\_DATE.patch

**Code Review Best Practices**:

**Motive and benefits**: Code reviews are important because they improve code quality and make your codebase more stable. In addition, they help programmers build relationships and work together more effectively.

**Education**: With code review, junior get advices from senior programmers within the context of their own code. There is no better way to learn the right practices and promote the company’s code guidelines and requirements.

**Communication**: With code review programmers necessarily talk to each other and share opinions. Doing so fosters companionship and improve the team cohesion.

**Knowledge spread**: Every team should struggle against code ownership i.e when one person owns a code base or a component. Code ownership is provoking friction and stress, both for the owner and for his/her colleagues. The owner doesn’t learn anymore from others and can become lazy since its work is not assessed by others. On the other hand the others can be let with a massive problem when the owner is absent or if he/she leaves the team. Code review is the best practice to spread knowledge and prevent code ownership.

Reviewing code with certain questions in mind can help you focus on the right things below are some key points:

Structure.

Style.

Logic.

Performance.

Test coverage.

Design.

Readability (and maintainability).

Functionality.

Do I understand what the code does?

Does the code function as I expect it to?

Does this code fulfill regulatory requirements?

**Give Feedback That Helps (Not Hurts)**

**If an egg is broken by outside force, life ends. If broken by inside force, life begins. Great things always begin from inside.**

Try to be constructive in your feedback, rather than critical. You can do this by asking questions, rather than making statements. And remember to give praise alongside your constructive feedback.

Giving feedback in-person (or even doing your review in-person) will help you communicate with the right tone.

Include Everyone in the Code Review Process

No matter how senior the programmer is, everyone needs to review and be reviewed. After all, everyone performs better when they know someone else will be looking at their work.

3. Foster a Positive Culture

Fostering a positive culture around reviews is important, as they play a vital role in product quality. It doesn’t matter who introduced the error. What matters is the bug was caught before it went into the product. And that should be celebrated.

**Checklist**:

1. Check branch is pushed on origin or not?
2. Check branch naming convention and also should mention in jira ticket.
3. Proper requirements mention in jira ticket with steps. Also should include that changes will go to APP portal or Lender portal or both
4. If its bug , then root cause of bug , and write some approach/solution, which are going to follow.
5. Impacts: This changes makes any impact on any existing functionality, if yes ,please mention all that.
6. Attached patch file should be updated with current date.
7. If any CSS or JS are changed , then versioning maintained or not in code, also there should not be **Inline** CSS/JS.
8. Does code moved on Dev branch or directly on stage branch?
9. Check if any existing function is reusable ,use it instead of creating new one.
10. There should not be present unnecessary **Print/echo /console.log() or exit()** in live code.
11. If any changes in DB , please verify tables data type , query optimization, unnecessary select \*, SQL escape and injection etc.
12. Function/Class or Variable naming convention: It should be clean , understandable and symmetric whole the function/file/project.
13. Proper comment : Please check whether comment is available for new function or code block, it should be self explained.
14. Check unnecessary For Loop within loop.
15. Please use Flag system in configuration to execute new implemented functionality to switch ON/OFF. If found any bug/breakage after production deployment, then switch accordingly without re-deployment whole codebase.