Table of Contents

[API Naming 2](#_Toc69838696)

[Logging Strategy 3](#_Toc69838697)

[Unit Testing Strategy 5](#_Toc69838698)

# API Naming

* **URIs as resources as nouns**

RESTful URIs should not indicate any kind of CRUD (Create, Read, Update, Delete) functionality. Instead, REST APIs should allow to manipulate a resource through one of the main HTTP methods.

Example: /users/{id} instead of /getUser

RESTful URIs should refer to a resource that is a thing (noun) instead of referring to an action (verb) because nouns have properties which verbs do not have – similar to resources have attributes.

* **Pluralized resources**

We should pluralize all resources unless they are singleton resources.

Example: /users (typical resource) or /users/{id}/address (singleton resource)

* **Forward slashes for hierarchy**

As shown in the examples above, forward slashes should be used to show the hierarchy between individual resources and collections.

Example: /users/{id}/address clearly falls under the /users/{id} resource which falls under the /users collection.

* **Punctuation for lists**

When there is no hierarchical relationship (such as in lists), punctuation marks such as the semicolon, or, more frequently, the comma should be used.

Example: /users/{id1},{id2} to access multiple user resources

* **Query parameters where necessary**

In order to [sort or filter](https://nordicapis.com/everything-you-need-to-know-about-api-pagination/) a collection, a REST API should allow query parameters to be passed in the URI.

Example: /users?location=USA to find all users living in the United States

* **Lowercase letters and dashes**

By convention, resource names should use exclusively lowercase letters. Similarly, dashes (-) should be used than that of underscores (\_).

Example: /users/{id}/pending-orders instead of /users/{id}/Pending\_Orders

* **Intuitive names (no jargon)**

Make your names intuitive! Wherever possible, choose the simplest and most commonly used word to refer to a given concept. Avoid using jargons.

Example: /users/{id}/card-number instead of /users/{id}/pan

* **No abridging**

Avoid abridging endpoint/resource names. Abridged names can actually create confusion in your API, as developers struggle to guess (and sometimes understand) the names you have chosen.

Example: /users/{id}/phone-number instead of /users/{id}/tel-no

* **No file extensions**

We souyld not use file extensions (such as .xml) out of your URIs. If you need to specify the format of the body, instead use the Content-Type header.

Example: /users/{id}/pending-orders instead of /users/{id}/pending-orders.xml

* **No trailing forward slash**

Do not add a trailing forward slash to the end of URIs.

Example: /users/{id}/pending-orders instead of /users/{id}/pending-orders/

* **Resource archetypes**

We should use uniform naming conventions across archetypes. For uniformity’s sake, resist the temptation to design resources that are hybrids of more than one archetype

**Document**

A document resource is a singular concept that is akin to an object instance or database record. In REST, you can view it as a single resource inside resource collection. A document’s state representation typically includes both fields with values and links to other related resources.

Use “singular” name to denote document resource archetype.

http://api.example.com/device-management/managed-devices/{device-id}

http://api.example.com/user-management/users/{id}

http://api.example.com/user-management/users/admin

**Collection**

A collection resource is a server-managed directory of resources. Clients may propose new resources to be added to a collection. However, it is up to the collection to choose to create a new resource or not. A collection resource chooses what it wants to contain and also decides the URIs of each contained resource.

Use the “plural” name to denote the collection resource archetype.

http://api.example.com/device-management/managed-devices

http://api.example.com/user-management/users

http://api.example.com/user-management/users/{id}/accounts

**Store**

A store is a client-managed resource repository. A store resource lets an API client put resources in, get them back out, and decide when to delete them. A store never generates new URIs. Instead, each stored resource has a URI. The URI was chosen by a client when it was initially put into the store.

Use “plural” name to denote store resource archetype.

http://api.example.com/song-management/users/{id}/playlists

**Controller**

A controller resource models a procedural concept. Controller resources are like executable functions, with parameters and return values; inputs and outputs.

Use “verb” to denote controller archetype.

http://api.example.com/cart-management/users/{id}/cart/checkout

http://api.example.com/song-management/users/{id}/playlist/play

# Logging Strategy

* **Logging Library**

All modules in Qualcomm should use SLF4j library for logging. The Spring Microservices can use The Spring Boot Starters all depend on spring-boot-starter-logging. This is where the majority of the logging dependencies for your application come from, including SLF4j.

* **Appenders**

Use rolling file appenders as specified in the attached log4j properties<TODO>.

* **Log Levels**

Think twice about a given log message. Not every bit of information is equally important and not every unexpected situation is an error or a critical message. In addition, using the logging levels consistently – information of a similar type should be on a similar severity level.

**TRACE** – the most fine-grained information only used in rare cases where you need the full visibility of what is happening in your application and inside the third-party libraries that you use. You can expect the TRACE logging level to be very verbose. You can use it for example to annotate each step in the algorithm or each individual query with parameters in your code.

**DEBUG** – less granular compared to the TRACE level, but it is more than you will need in everyday use. The DEBUG log level should be used for information that may be needed for diagnosing issues and troubleshooting or when running application in the test environment for the purpose of making sure everything is running correctly.

INFO – the standard log level indicating that something happened, application entered a certain state, etc. For example, a controller of your authorization API may include an INFO log level with information on which user requested authorization if the authorization was successful or not. The information logged using the INFO log level should be purely informative and not looking into them on a regular basis shouldn’t result in missing any important information.

**WARN** – the log level that indicates that something unexpected happened in the application, a problem, or a situation that might disturb one of the processes. But that doesn’t mean that the application failed. The WARN level should be used in situations that are unexpected, but the code can continue the work. For example, a parsing error occurred that resulted in a certain document not being processed.

**ERROR** – the log level that should be used when the application hits an issue preventing one or more functionalities from properly functioning. The ERROR log level can be used when one of the payment systems is not available, but there is still the option to check out the basket in the e-commerce application or when your social media logging option is not working for some reason.

FATAL – the log level that tells that the application encountered an event or entered a state in which one of the crucial business functionality is no longer working. A FATAL log level may be used when the application is not able to connect to a crucial data store like a database or all the payment systems are not available and users can’t checkout their baskets in your e-commerce.

* **Log Format**

The log format should be consistent across all modules. All microservices should follow the same pattern while logging. Attached the logging template<TODO>

* **Filters and Aspects**

Adding log statements to the start and end of every method is messy. The better way is to use filters and aspects and avoid the code duplication.

**Filter Vs Aspect-Oriented Programming -**

If you are looking to create log statements related to specific requests, you should opt for using filters, as they are part of the handling chain that your application already goes through for each request. They are easier to write, easier to test and usually more performant than using aspects. If you are considering more cross-cutting concerns, for example, audit logging, or logging every method that causes an exception to be thrown, use AOP.

Filters can be registered with your web container by creating a class implementing javax.servlet.Filter and annotating it with @Component, or adding it as an @Bean in one of your configuration classes. When your spring-boot-starter application starts up, it will create the Filter and register it with the container.

You can choose to create your own Filter, or to [use an existing one](https://github.com/spring-projects/spring-framework/blob/master/spring-web/src/main/java/org/springframework/web/filter/CommonsRequestLoggingFilter.java). To make use of the existing Filter, [you need to supply a CommonsRequestLoggingFilter bean](https://gist.github.com/NutterzUK/cf3e9ff02b17454cf2a5bc28610c051c) and set your logging level to debug.

**Aspects for Cross-Cutting Concerns -** [Aspect-oriented programming](https://docs.spring.io/spring-framework/docs/4.3.15.RELEASE/spring-framework-reference/html/aop.html) enables you to fulfill cross-cutting concerns, in one place. You can do this without your logging code needing to sprawl across every class. All APIs should use AOP to log the below cases.

Any exceptions thrown from any method within your packages (See @AfterThrowing)

Performance metrics by timing before/after each method is run (See @Around)

Audit logging (use custom annotation) for all critical transactions

* **Mapped Diagnostic Context**

We should be able to track all requests coming to the APIs separately. Use MDC and Follow the below steps to track the individual requests.

Add a header (‘tracking-id’) to each request going to your API. You can generate this on the fly (use UUID).

Create a filter that runs once per request and stores that value in the MDC.

Update your logging pattern to reference the key in the MDC to retrieve the value.

Make sure to clear up after the request by calling MDC.clear(), preferably in a finally block so that it always runs.

After setting the value on your MDC, just add %X{tracking}  to your logging pattern (Replacing the word “tracking” with the key you have put in MDC) and your logs will contain the value in every log message for that request.

* **Avoid Logging Sensitive Information**

Sensitive information like passwords, credit card numbers, social security numbers, and access tokens should not be added in logs or should be masked. It will be dangerous if leaked or accessed by those who should not see that.

# Unit Testing Strategy

All Java modules should use Junit for unit testing. All modules should have >90% coverage. The repository, config, model packages can be ignored for unit testing. Developers should take care of the below standards while writing unit test cases. Developers can use json as data provider for Mockito, whenever necessary<TODO>.

* All methods, regardless of visibility, should have appropriate unit tests.
* There should be exactly one assertion per unit test cases.
* Do not use static members in a test class. If you have used then re-initialize for each test case.
* Do not write your own catch blocks that exist only to fail a test.
* Do not print anything out in unit tests.
* Team should Mock out all external services and state