UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

MODULO TESTING

Testing dei progetti Apache
BookKeeper e Syncope

Purificato Marco
[**0339286**]

A.A 2022-2023

INTRODUZIONE

Il presente report si pone l'obiettivo di andare a descrivere in maniera dettagliata, l'applicazione delle tecniche e degli strumenti descritti nel modulo di software testing del corso su due progetti open-source, appartenenti all'Apache Software Foundation, di cui i sorgenti reperibili su GitHub: BookKeeper e Syncope.

Tramite l'utilizzo di un approccio black-box basato sulle funzionalità attese, quindi andando per quanto possibile a reperire le informazioni necessarie alla stesura dei test, tramite la relativa documentazione, si è proceduto con la rimozione di tutti i test già presenti e con l'implementazione dei test di *unità* ed *integrazione* tramite il framework JUnit 4 che permettere di rendere testabile il processo di build, parallelamente al framework Mockito [ver. BookKeeper{3.12.4} – Syncope{5.3.1}] che permette di simulare parte dell'ambiente di esecuzione del test.

Per orchestrare l'esecuzione dei test come parte del processo di build automatizzato si sono utilizzati i plugin Surefire [ver. BookKeeper{2.22.2} - Syncope{3.1.0}] e Filesafe [ver. 3.0.0-M4], rispettivamente per i test di unità ed i test di integrazione.

Conseguentemente alla stesura dei test, è stato utilizzato per la misurazione del code-coverage e quindi per l'analisi di adeguatezza basata sul control-flow il tool JaCoCo [ver. BookKeeper{0.8.8} - Syncope{0.8.9}], Ba-Dua per la misurazione del data-flow coverage ed infine Pitest (aka PIT) [ver. BookKeeper{1.5.1} - Syncope{1.14.1}] per quanto riguarda il mutation coverage sulle classi di test sviluppate.

AMBIENTE

Prendendo in considerazione la *Software Quality Assurance*, in particolare le componenti del Software Configuration Managment e del Controllo Qualità; lo sviluppo dei test è avvenuto tramite l'utilizzo di Git come sistema automatizzato di gestione delle versioni, Maven [ver. 3.9.1] come sistema di build e GitHub Actions come software di Continuos Integration (CI), le build sono state effettuate localmente su Windows 11 Pro 22H2 e in remoto tramite GitHub Actions su macchine virtuali aventi l'ultima versione di Ubuntu disponibile la 22.04, utilizzando per BookKeeper la versione di Java 8, scelta perché unica che ha permesso la piena compatibilità tra i vari tool e per Syncope la versione 17; in particolare all'interno della directory `.github/workflows` si hanno i seguenti file YAML di configurazione:

> `badua-flow.yml` : esegue il profilo relativo a Ba-Dua e procede con l'upload del report;
>
> `jacoco-flow.yml` : esegue il profilo relatvo a JaCoCo e procede con l'upload del report;
>
> `pit-flow.yml` : esegue il profilo relatvo a Pitest e procede con l'upload del report;
>
> `maven-flow.yml` : esegue la fase di `verify`, quindi esegue eventuali controlli per verificare la validità, e successivamente procede con l'upload dei report relativi a surefire e failsafe.

Per l'esecuzione di ognuno dei tool di coverage è stato creato un Profilo di Build Maven dedicato, in particolar modo per quanto riguarda Ba-Dua oltre al profilo, non essendo distribuito con Maven Central, per usare le sue risorse si è effettuata un'istallazione locale del tool alla versione 0.6.0, così da poter riferire nel progetto oltre che le dipendenze necessarie, il file ba-dua-cli-0.6.0-all.jar incaricato alla conversione dell'oggetto badua.ser, nel file di report in formato .xml, presente nella directory aggiuntiva badua-data-coverage; così risolvendo i problemi di compatibilità dovuti al plugin exec-maven-plugin. In entrambi i progetti, per ogni tool è possibile richiamare i vari profili Maven, tramite i seguenti comandi:

JaCoCo – Control Flow Coverage

```
mvn clean verify -P jacoco
```

Ba-Dua – Data Flow Coverage

```
mvn clean verify -P badua
```

Pitest – Mutation Coverage

```
mvn clean test -P pit
```

BOOKKEEPER

BookKeeper è un sistema di registrazione distribuito, progettato per gestire la memorizzazione affidabile di log, spesso utilizzato per implementare funzionalità di registrazione, replicazione e persistenza in applicazioni distribuite. In questo sistema il concetto chiave è il "ledger". Un ledger è un registro distribuito che contiene un insieme di voci ordinate sequenzialmente. Ciascuna voce nel ledger ha un numero di sequenza univoco, e i dati associati a ciascuna voce vengono memorizzati in modo affidabile su diversi nodi del cluster BookKeeper detti bookie.

Proprio per questo suo ruolo cardine, oltre alla presenza di documentazione che ci permette di perseguire un approccio il più possibile black-box, le classi da testare scelte sono state:

`org.apache.bookkeeper.client.BookKeeper`

`org.apache.bookkeeper.bookie.storage.ldb.ReadCache`

BookKeeper.java

La classe BookKeeper è un'importante componente nell'ecosistema di Apache BookKeeper dato il suo ruolo di gestione dei ledger, come per esempio creazione, controllo di conformità, apertura, scrittura ed eliminazione. I metodi della classe testati sono i seguenti:

- `public LedgerHandle createLedger(int ensSize, int writeQuorumSize, int ackQuorumSize, DigestType digestType, byte[] passwd, final Map<String, byte[]> customMetadata)`
- `public void deleteLedger(long lId)`
- `public LedgerHandle openLedger(long lId, DigestType digestType, byte[] passwd)`

Per poter manipolare i ledger e quindi poter testare le classi appena elencante, nei test si è dovuto estendere la classe *BookKeeperClusterTestCase* mantenendo alcune classi dipendenti, queste classi di test preesistenti nel progetto e non eliminate nella fase di pulizia dei test citata nell'introduzione, sono state mantenute perché ci permettono di inizializzare un cluster di nodi Bookie, con cui far interagire i ledger, e ci permettono di istanziare Zookkeeper per il coordinamento dei nodi e la gestione dei metadati dei ledger. In questo caso non è stato utilizzato Mockito per la simulazione di questo ambiente di esecuzione, perché si è voluto fare testing su di un ambiente più vicino possibile ad uno reale, estremamente dinamico, dinamicità però supportata da una scarsa documentazione che ci avrebbe portato ad avvalerci di un approccio troppo white-box per lo sviluppo dei mock; quindi ci avrebbe portato ad un approccio non basato sulle funzionalità attese ma al

contrario, basato sulla specifica direttamente eseguibile sul SUT, nel nostro caso il codice e tutto quello concerne ai cluster di Bookie.

Il metodo **createLedger(..)** hai il compito, come indicato anche dal nome, di creare un nuovo ledger tramite il passaggio dei seguenti parametri:

- ensSize, il numero di nodi su cui verrà memorizzato il ledger;
- writeQuorumSize, il numero di nodi dove ciascuna entry verrà memorizzata, quindi il valore massimo di replicazione di ciascuna entry;
- ackQuorumSize, il numero di nodi da cui si aspetta un acknowledgement post scrittura;
- digestType, il tipo di digest utilizzato per aprire i ledger, impostato CRC32 in default ma selezionabili altri formati: MAC, CRC32C, DUMMY ;
- passwd, la password, una sequenza di byte utilizzata per aprire il ledger;
- customMetadata, metadata custom, composti da oggetti complessi formati da coppie chiave – valore: < stringa – sequenza di byte >.

I domini di input sono stati identificati grazie alla documentazione presente sul sito ufficiale del progetto [*Link1* e *Link2*], che ci specifica anche una relazione obbligatoria che se non rispettata porterà al fallimento della creazione del ledger, la relazione è la seguente:

ensSize >= writeQuorumSize >= ackQuorumSize

Successivamente si è proceduto alla partizione dei domini di input in gruppi che possono essere considerati comuni, ricavando per ciascun parametro le seguenti classi di equivalenza:

- enSize: {<=0}, {>0}

- writeQuorumSize: {<=EnSize}, {>EnSize}

- ackQuorumSize: {<=WriteQuorumSize}, {> WriteQuorumSize }

- digestType: {MAC}, {CRC32}, {CRC32C}, {DUMMY}

E secondo i criteri guida di partizionamento per i dati di tipo array e di tipo complesso:

- passwd: null, {“empty”}, {“no_empty”}, {“exceed_dimension”}

- customMetadata: null, {“valid_instance”}, {“invalid_instance”}

Dato che empiricamente si è dimostrato come i problemi più comuni avvengano sugli elementi che si trovano lungo i bordi delle classi di equivalenza si è proceduto con l’analisi dei boundary values:

- enSize: -1; 0; 1;

- writeQuorumSize: EnSize-1; EnSize; EnSize+1;

- ackQuorumSize: WriteQuorumSize-1; WriteQuorumSize; WriteQuorumSize+1;

- digestType: MAC; CRC32; CRC32C; DUMMY;

- passwd: null; {“empty”}; {“no_empty”}; Integer.MAX_VALUE; Integer.MAX_VALUE-8;

- customMetadata: null; {“empty”}; {“valid_instance”}; {“invalid_instance”};

Prendendo in considerazione le classi di equivalenza del parametro passwd, in accordo alla documentazione di Java [*Link_3*], per ottenere un array di byte che eccedesse la dimensione consentita supportata, lo si è inizializzato con dimensione pari alla costante `Integer.MAX_VALUE` che rappresenta la dimensione massima che può assumere un intero ergo lo si è inizializzato con una

dimensione pari a 2^31 = 2,147,483,648. Dato che un array per la sua inizializzazione ha bisogno di 8 byte dedicati, si è passato anche il valore a bordo ammissibile `Integer.MAX_VALUE-8`.

Analizzando il parametro customMetadata e rispettando le linee guida di partizionamento per un dato di tipo complesso, si è aggiunta tra le sue classi di equivalenza: {“invalid_instance”}; anche se il costruttore della classe che implementa l’interfaccia Map torna sempre istanze corrette perché non permette il passaggio di parametri diversi da quelli dichiarati <String, byte[]>, nulla vieta che il suo codice potrebbe variare introducendo bug, proprio per questo si è creata una nuova versione della classe HashMap chiamata TestNotMap che implementa l’interfaccia Map ma ritorna, dopo aver fornito la chiave, un valore intero anzi che una stringa.

Per combinare i parametri provenienti dalle differenti classi di equivalenza, per poi passarli in input ai test, si è optato l’uso di entrambi gli approcci disponibili. Approccio multidimensionale per i parametri enSize (**enS**) – writeQuorumSize (**wQs**) – ackQuorumSize (**aQs**), un approccio più reliable anche se più costoso essendo combinazione lineare di tutte le classi di equivalenza su tutti i parametri, costo non impattante in questo caso dato che si parla di una combinazione di dati esigua e quindi non particolarmente impattante in termini di complessità sul sistema; ed un approccio unidimensionale per i parametri digestType (**digType**), passwd e customMetadata (**custmMetadt**).

#N.	enS	wQs	aQs	digType	passwd	custmMetadt	OUTPUT
0	-1	-2	-3	CRC32	"p@SSw0rd".getBytes()	null	IllegalArgumentException
1	-1	-2	-2	CRC32	"p@SSw0rd".getBytes()	cstmMetdataNotValid	IllegalArgumentException
2	-1	-2	-1	CRC32	"p@SSw0rd".getBytes()	cstmMetadata	IllegalArgumentException
3	-1	-1	-2	MAC	"p@SSw0rd".getBytes()	cstmMetadata	IllegalArgumentException
4	-1	-1	-1	DUMMY	new byte[]{}	cstmMetadata	IllegalArgumentException
5	-1	-1	0	MAC	null	cstmMetadata	IllegalArgumentException
6	-1	0	-1	CRC32C	"p@SSw0rd".getBytes()	cstmMetdataNotValid	IllegalArgumentException
7	-1	0	0	CRC32	byte[Integer.MAX_VALUE]	Collections.emptyMap()	OutOfMemoryError
8	-1	0	1	CRC32	byte[Integer.MAX_VALUE-8]	cstmMetadata	IllegalArgumentException
9	0	-1	-2	CRC32C	"p@SSw0rd".getBytes()	Collections.emptyMap()	CREATO CON SUCCESSO
10	0	-1	-1	CRC32	"p@SSw0rd".getBytes()	cstmMetdataNotValid	NullPointerException
11	0	-1	0	DUMMY	"p@SSw0rd".getBytes()	cstmMetadata	IllegalArgumentException
12	0	0	-1	MAC	"p@SSw0rd".getBytes()	cstmMetadata	CREATO CON SUCCESSO
13	0	0	0	MAC	new byte[]{}	cstmMetadata	IllegalArgumentException
14	0	0	1	MAC	null	Collections.emptyMap()	IllegalArgumentException
15	0	1	0	CRC32C	"p@SSw0rd".getBytes()	cstmMetdataNotValid	NullPointerException
16	0	1	1	CRC32	null	cstmMetadata	NullPointerException
17	0	1	2	MAC	new byte[]{}	cstmMetadata	IllegalArgumentException

18	1	0	-1	MAC	new byte[]{}	cstmMetadata	CREATO CON SUCCESSO
19	1	0	0	CRC32	new byte[]{}	cstmMetadata	CREATO CON SUCCESSO
20	1	0	1	CRC32	Null	cstmMetadata	IllegalArgumentException
21	1	1	0	CRC32	"p@SSw0rd".getBytes()	null	CREATO CON SUCCESSO
22	1	1	1	CRC32	"p@SSw0rd".getBytes()	cstmMetdataNotValid	NullPointerException
23	1	1	2	CRC32C	"p@SSw0rd".getBytes()	cstmMetadata	IllegalArgumentException
24	1	2	1	DUMMY	"p@SSw0rd".getBytes()	cstmMetadata	TestTimedOutException
25	1	2	2	DUMMY	new byte[]{}	cstmMetadata	TestTimedOutException
26	1	2	3	DUMMY	null	Collections.emptyMap()	IllegalArgumentException

Dai risultati del test si evince che, come atteso, non rispettando la relazione vincolante che lega i parametri { ensSize >= writeQuorumSize >= ackQuorumSize }, si scaturisce un eccezione del tipo `java.lang.IllegalArgumentException`; che passando per la creazione del ledger metadati non validi "cstmMetdataNotValid" oppure una password `null`, si scaturisce un eccezione del tipo `java.lang.NullPointerException` e che inizializzando una password di una dimensione superiore a quella consentita (configurazione 7) si scaturì un eccezione del tipo `org.junit.runners.model.TestTimedOutException`.

Pur rispettando, invece, la relazione, unico vincolo esposto nella documentazione, esistono dei casi anomali che comunque scaturiscono un'eccezione, come un valore negativo del parametro enSize (configurazione 0, 3 e 5), quindi il numero di nodi dove verrà memorizzato il ledger non può essere negativo; e la terna (0,0,0), quindi non ci può essere un ledger che si interfaccia in totale con 0 nodi.

Inoltre si è notato come, la configurazione 8, sia un test ammissibile per la macchina locale ma non per le macchine virtuali fornite da GitHub Action che restituiscono un eccezione del tipo `OutOfMemoryError`, per completare le build remote, si è commentata la configurazione.

Altre configurazioni anomale sono la 24 e 25 dove il sistema, a differenza dei casi precedenti che a seguito del controllo della relazione non rispettata dei parametri enS/wQs/aQs ritornano un'eccezione del tipo `IllegalArgumentException`, entra in un ciclo di creazione infinito fino al sopraggiungere del time-out del test `org.junit.runners.model.TestTimedOutException`. Dato che al sopraggiungere del time-out o dell'eccezione `OutOfMemoryError` la build non può procedere, queste quattro configurazioni sono state commentate.

Il metodo **openLedger(..)** ha il compito di aprire un determinato ledger riferito con il suo Id passato come parametro, tramite una determinata password anch'essa passata in input. I domini di input sono stati partizionati come segue:

- lId: {<=0},{>0}
- digestType: {MAC}, {CRC32}, {CRC32C}, {DUMMY}
- passwd: null, {"empty"}, {"no_empty"}, {"exceed_dimension"}

Essendo, da documentazione, l'id di un ledger un intero a 64bit univoco nel sistema, un id non valido non sarà altro che un id di un ledger che non è presente tra i ledger già esistenti nel cluster di Bookie;

quindi, in fase di configurazione del test è stato preventivamente inserito un ledger nel cluster. I parametri passati, quindi, tramite un approccio unidimensionale, sono stati i seguenti:

#N	lId	digestType	passwd	OUTPUT
0	0	CRC32	"p@SSw0rd".getBytes()	APERTO CON SUCCESSO
1	-1	DUMMY	null	BKException$BKNoSuchLedgerExistsOnMetadataServerException
2	0	CRC32	"n0t-p@SSw0rd".getBytes()	BKException$BKUnauthorizedAccessException
3	1	CRC32C	new byte[]{}	BKException$BKNoSuchLedgerExistsOnMetadataServerException
4	0	MAC	byte[Integer.MAX_VALUE]	OutOfMemoryError
5	0	MAC	byte[Integer.MAX_VALUE-8]	BKException$BKUnauthorizedAccessException

Dai risultati dei test si è notato come l'inserimento di un password di dimensione pari a `Integer.MAX_VALUE-8` oppure non valida, quindi non corrispondente a quella usata per la creazione del ledger che si sta tentando di aprire, generi un eccezione del tipo `org.apache.bookkeeper.client.BKException$BKUnauthorizedAccessException`.

Anche se la documentazione riporta come l'id un numero intero generico di tipo long quindi un numero intero appartenente all'intervallo [-2^{63}, 2^{63}– 1], durante la fase di test si è notato come passando un id negativo questo, a priori dal check della sua corrispondenza all'interno del Bookie, scaturisca un'eccezione del tipo `org.apache.bookkeeper.client.BKException$BKNoSuchLedgerExistsOnMetadataServerException`.

Il metodo **deleteLedger(..)** ha il compito di eliminare un ledger avente un determinato id passato come parametro, dove è stato partizionato come segue:

- lId: {<=0},{>0}

Come nel test del metodo precedente, in fase di configurazione del test è stato preventivamente inserito un ledger nel cluster. Di seguito i parametri passati in input:

#N	lId	OUTPUT
0	0	ELIMINATO CON SUCCESSO
1	-1	NON ELIMINATO
2	1	NON ELIMINATO

Dall'esecuzione dei test si evince che, a differenza dei test precedenti l'eliminazione di un id non valido, quindi negativo o non esistente, non scaturisce un'eccezione, ma semplicemente non viene eseguita nessuna operazione di eliminazione; proprio per questo il controllo della mancata o corretta eliminazione è stato fatto tramite l'utilizzo della classe BookKeeperAdmin, che fornisce metodi di gestione e controllo dei cluster di Bookie.

ReadCache.java

La classe ReadCache implementa un meccanismo di caching per i dati in lettura, uno spazio di memoria suddiviso in diversi segmenti, che viene utilizzato per migliorare le prestazioni delle operazioni di lettura sui ledger. I metodi della classe testati sono i seguenti:

- `public void put(long ledgerId, long entryId, ByteBuf entry)`
- `public ByteBuf get(long ledgerId, long entryId)`
- `public boolean hasEntry(long ledgerId, long entryId)`

Il metodo **put(..)** ha il compito di scrivere nella cache il contenuto di tipo ByteBuf passato come parametro, questo contenuto sarà poi scritto in una determinata entry di un determinato ledger, entrambi caratterizzati con un proprio id, passati anch'essi come parametri.

Dalla documentazione ufficiale si possono studiare questi tre parametri, ricorrenti a tutti e tre i metodi testati; ledgerId ed entryId sono dei numeri interi univoci a 64bit, a differenza di entry che è un oggetto di tipo ByteBuf.

Un oggetto di tipo ByteBuff è un array di byte ad accesso sequenziale o randomico che, a differenza di un semplice array di byte (byte[]), offre dei meccanismi di gestione più efficienti [*Link 4*]; per il suo partizionamento, infatti, si è usato il criterio guida per un dato di tipo array.

Dall'analisi dei parametri possiamo, quindi, procedere con il seguente partizionamento dei domini di input:

- ledgerId: {<=0}, {>0}

- entryId: {<=0}, {>0}

- entry: null, {"empty"}, {"no_empty"}, {"exceed_dimension"}

Con la conseguente analisi dei valori a bordo delle classi di equivalenza:

- ledgerId: -1; 0; 1;

- entryId: -1; 0; 1;

- entry: null; {"empty"}; {"no_empty"}; (CACHE_CAPACITY+1);

Dove, con "(CACHE_CAPACITY+1)" si intende una entry con dimensione superiore alla dimensione dell'intera cache che la ospita.

I parametri passati in input, in accordo ad un approccio multidimensionale per gli Id ed unidimensionale per la entry, sono i seguenti:

#N	ledgerId	entryId	entry	OUTPUT
0	-1	-1	valid_entry	IllegalArgumentException
1	-1	0	valid_entry	IllegalArgumentException
2	-1	1	valid_entry	IllegalArgumentException
3	0	-1	valid_entry	SCRITTO CON SUCCESSO
4	0	0	null	NullPointerException
5	0	1	exceedDimension_entry	SCRITTURA NON ESEGUITA
6	1	-1	valid_entry	SCRITTO CON SUCCESSO
7	1	0	null	NullPointerException
8	1	1	valid_entry	SCRITTO CON SUCCESSO

Dai risultati del test si evince che, se pur dello stesso tipo (long) e documentati in maniera speculare l'id di una entry può assumere valori negativi, infatti nei casi in cui il test scaturisce un eccezione del tipo `java.lang.IllegalArgumentException`, questo è dovuto unicamente a ledgerId. Inoltre, si può notare che, come output atteso al passaggio di un entry `null`, venga scaturita un'eccezione del tipo `NullPointerException` e che in maniera non attesa, lo scenario in cui la dimensione di una entry è maggiore della dimensione della cache sia nativamente gestista, infatti non scaturisce nessuna exception, ma si limiterà al fare il logging del Warning e non inserire l'entry in cache.

Il metodo **get(..)** restituisce, se presente in cache, il contenuto di tipo ByteBuf di una determinata entry associata ad un determinato ledger, entrambi identificati da un Id passato come parametro. Di seguito il partizionamento dei domini di input:

- ledgerId: {<=0}, {>0}

- entryId: {<=0}, {>0}

La lettura nella cache avviene con successo quando si prova ad effettuare la lettura passando degli id validi; quindi, degli id riferiti in una delle entry già presente in cache, per simulare lo scenario, in fase di configurazione dei test viene inserita preventivamente nella cache una entry valida.

I parametri passati in input, in accordo ad un approccio multidimensionale, sono i seguenti:

#N	ledgerId	entryId	OUTPUT
0	-1	-1	IllegalArgumentException
1	-1	0	IllegalArgumentException
2	-1	1	IllegalArgumentException
3	0	-1	null
4	0	0	null
5	0	1	null
6	1	1	LETTO CON SUCCESSO
7	1	0	null
8	1	-1	null

Eseguendo i test si può notare come lo scenario di leggere una entry non esistente nella cache sia gestita nativamente nell'implementazione del metodo ritornando un oggetto `null`, e come l'unica eccezione che si possa scaturire tramite i parametri passati sia quella dovuta ad un id del ledger negativo.

Il metodo **hasEntry(..)** restituisce il valore booleano *true* se è presente nella cache una determinata entry di un determinato ledger, identificati tramite id passati come parametri, altrimenti restituisce `false`, indicando al sistema di andare a prelevare i dati, se disponibile, nello spazio di archiviazione in persistenza.

L'analisi dei domini di input con il relativo partizionamento ed analisi ai bordi delle classi di equivalenza è la medesima del metodo precedente.

- ledgerId: {<=0}, {>0}

- entryId: {<=0}, {>0}

I parametri passati in input, in accordo ad un approccio multidimensionale, sono i seguenti:

#N	ledgerId	entryId	OUTPUT
0	-1	-1	IllegalArgumentException
1	-1	0	IllegalArgumentException
2	-1	1	IllegalArgumentException
3	0	-1	false
4	0	0	false
5	0	1	false
6	1	1	LETTO CON SUCCESSO
7	1	0	false
8	1	-1	false

Come per il metodo precedente, si può notare come lo scenario di leggere una entry non esistente nella cache sia gestita nativamente nell'implementazione del metodo, ritornando un oggetto booleano settato a `false`, e come l'unica eccezione che si possa scaturire tramite i parametri passati sia quella dovuta ad un id del ledger negativo.

ADEGUATEZZA DEI TEST e MIGLIORAMENTO

Di seguito si andranno ad analizzare tramite tre diversi criteri di copertura, l'adeguatezza e la bontà dei test sviluppati, due basati sul control-flow graph che descrivono le relazioni di dipendenza tra le varie istruzioni del SUT, tramite l'utilizzo del tool JaCoCo ed uno basato sul data-flow graph che descrive l'uso di variabili/dati all'interno del SUT, tramite l'utilizzo del tool Ba-Dua.

JACOCO

Per quanto riguarda i criteri basati sul control-flow graph di seguito viene riportata l'analisi del branch-coverage e dello statement-coverage effettuata tramite il tool JaCoCo.

La classe *BookKeeper* presenta uno statement-coverage del 38% ed un branch-coverage del 21%, in particolare:

- il metodo `public LedgerHandle createLedger(int ensSize, int writeQuorumSize, int ackQuorumSize, DigestType digestType, byte[] passwd, final Map<String, byte[]> customMetadata)` è caratterizzato da uno statement-coverage dell'81% e di un branch-coverage del 50%. Non viene raggiunto il branch dovuto al ritorno di un handle nullo di un ledger [*Figura 1*];

- il metodo `public LedgerHandle createLedger(int ensSize, int writeQuorumSize, int ackQuorumSize, DigestType digestType, byte[] passwd)` è caratterizzato da uno statement-coverage del 100% e di un branch-coverage indefinito [*Figura 2*];

- il metodo `public void deleteLedger(long lId)` è caratterizzato da uno statement-coverage del 100% e di un branch-coverage indefinito [*Figura 3*];

- il metodo `public LedgerHandle openLedger(long lId, DigestType digestType, byte[] passwd)` è caratterizzato da uno statement-coverage del 100% e di un branch-coverage indefinito [*Figura 4*].

La classe *ReadCache* presenta uno statement-coverage del 81% ed un branch-coverage del 83%, in particolare:

- il metodo `public void put(long ledgerId, long entryId, ByteBuf entry)` è caratterizzato da uno statement-coverage del 48% e di un branch-coverage del 50%. Non viene raggiunto il branch dove (`offset + entrySize > segmentSize`) nel caso in cui non si possa inserire l'entry in un segmento [*Figura 5*];

- il metodo `public ByteBuf get(long ledgerId, long entryId)` è caratterizzato da uno statement-coverage del 100% e di un branch-coverage del 100% [*Figura 6*];

- il metodo `public boolean hasEntry(long ledgerId, long entryId)` è caratterizzato da uno statement-coverage del 100% e di un branch-coverage del 100% [*Figura 7*].

BA-DUA

Dato che gli approcci white-box basati sul control-flow graph hanno lo svantaggio, per loro natura, di non riuscire a rilevare le funzionalità mancanti, si procede con un approccio basato sul data-flow graph tramite il tool Ba-Dua; quindi, un approccio basato sulla definizione ed uso di dati, che essendo

distribuiti nell'applicazione, risulta molto improbabile che vengano eliminati tutti i loro relativi record dal SUT.

Per quanto riguarda la classe *BookKeeper*:

nel metodo `createLedger`, come confermato nell'analisi precedente, viene mancato l'uso del LOG definito a riga 952, che corrisponde al parametro def="962" all'interno del report, scaturito dalla creazione di un ledger nullo [*Figura 8*];

per i metodi `deleteLedger` e `openLedger`, invece, non troviamo informazioni relative all'interno del report xml generato.

Per quanto riguarda la classe *ReadCache*, a differenza della classe precedente, troviamo un numero di utilizzazioni dei dati molto più alto:

nel metodo put viene mancato l'utilizzo della variabile `entrySize` ed `alignedSize` nella condizione in cui (`offset + entrySize > segmentSize`), e delle variabili `ledgerId`, `entryId`, `offset` e `entrySize`, dopo riga 117, parte di codice raggiungibile solo nel caso non si possa inserire una entry nel segmento [*Figura 9*];

nel metodo `get` si hanno coppie def-use coperte 31 e mancate 1;

nel metodo `hasEntry` si hanno coppie def-use coperte 22 e mancate 1.

MIGLIORAMENTO DEI TEST

A seguito delle precedenti analisi, di seguito si cerca di modificare i test in modo tale da aumentare l'adeguatezza e quindi la loro percentuale di coverage ove non raggiunto il 100%.

Prendendo in considerazione la classe *BookKeeper*, per aumentare la coverage e quindi l'adeguatezza dei test implementati, bisogna intervenire sul test del metodo `createLedger(int ensSize, int writeQuorumSize, int ackQuorumSize, DigestType digestType, byte[] passwd, final Map<String, byte[]> customMetadata)`, essendo l'unico nella classe con coverage al di sotto del 100%; per fare questo bisogna raggiungere l'unico branch mancato dovuto al ritorno di un handle nullo. Andando ad analizzare il codice si è osservato che un handle nullo può essere risultato solo di un time-out definito a 292 anni, scenario che non si è riuscito a replicare, se non tramite un mock del metodo statico `waitForResult()` della classe `SyncCallbackUtils`, l'unico problema è stato quello di non poter eseguire correttamente il metodo di test sviluppato `nullHandleMockedScenarioTest()` a causa di un incompatibilità tra la versione java richiesta per l'inizializzazione di un cluster di bookie, necessario per la creazione di un ledger, Java 8, e la versione minima di Mockito, la 5.0.0, che introduce il mock di metodi statici e che richiede però, una versione superiore a Java 8; l'implementazione di quest'ultimo test con il mock statico è stato quindi annotato con `@Ignore`.

Per aumentare la coverage della classe, quindi, si è proceduto con la creazione di un nuovo set di test che prendono in considerazioni le restanti varianti non testate del metodo `createLedger` ed il metodo `createLedgerAdv`, utilizzando oltretutto una nuova configurazione del client detta `BookieHealthCheck`, che permette un controllo sull'integrità dei bookie ed allo stesso tempo il raggiungimento di un nuovo branch. Riguardo le varianti del metodo `createLedger`, essendo invocazioni del metodo principale ampiamente testato ed analizzato in precedenza, ma con il passaggio di meno parametri, dove quelli non passati vengono assegnati staticamente, per non portare ridondanza all'interno del report, non si riprocede all'analisi dei domini di input con il loro relativo partizionamento e l'analisi scaturita dai set di input; ed essendo i medesimi si procede con un approccio unidimensionale.

Unico set di input non testato precedentemente nell'analsi di `createLedger`, che troviamo codificato staticamente [*Figura 9*] passando solo la password ed il digestType al metodo, è il seguente:

#N.	enS	wQs	aQs	digType	passwd	custmMetadt	OUTPUT
0	3	2	2	CRC32C	"p@SSw0rd".getBytes()	Collections.emptyMap()	BKException$BKNotEnoughBookiesException
1	3	2	2	CRC32	byte[Integer.MAX_VALUE]	Collections.emptyMap()	OutOfMemoryError
2	3	2	2	CRC32	byte[Integer.MAX_VALUE-8]	Collections.emptyMap()	IllegalArgumentException / TestTimedOutException
3	3	2	2	MAC	new byte[]{}	Collections.emptyMap()	BKException$BKNotEnoughBookiesException
4	3	2	2	DUMMY	"p@SSw0rd".getBytes()	Collections.emptyMap()	BKException$BKNotEnoughBookiesException

Dai risultati si osserva che nel caso in cui viene passata la configurazione 2 al metodo `public LedgerHandle createLedger(DigestType digestType, byte[] passwd)`, anzi che scaturire come in tutte le varianti di `createLedger()` testate un eccezione del tipo `IllegalArgumentException`, entra in un ciclo anomalo di creazione infinito fino al sopraggiungere del time-out del test `org.junit.runners.model.TestTimedOutException`. Dato che al sopraggiungere del time-out la build non può procedere, tramite l'utilizzo del costrutto `Assume.assumeTrue()` fornito da JUnit, questo scenario di test viene ignorato.

Oltretutto, sviluppando i test nei casi precedenti si sono presi in considerazione i valori ai bordi con le loro relazioni tra le varie classi di equivalenza; così facendo, però, non si era andati in contro ad uno scenario dove `enSize` fosse maggiore del numero di bookie impostato da creare durante il setup del test, cioè maggiore di 2.

Andando a passare un `enSize` maggiore del numero di bookie effettivamente creati nel cluster si scaturirà la seguente eccezione:

```
org.apache.bookkeeper.client.BKException$BKNotEnoughBookiesException
```

Per risolvere questa eccezione ed eseguire correttamente i test, si è inizializzato il numero di bookie in numero ≥ 3, producendo i seguenti risultati:

#N.	enS	wQs	aQs	digType	passwd	custmMetadt	OUTPUT
0	3	2	2	CRC32C	"p@SSw0rd".getBytes()	Collections.emptyMap()	CREATO CON SUCCESSO
1	3	2	2	CRC32	byte[Integer.MAX_VALUE]	Collections.emptyMap()	OutOfMemoryError
2	3	2	2	CRC32	byte[Integer.MAX_VALUE-8]	Collections.emptyMap()	IllegalArgumentException / TestTimedOutException
3	3	2	2	MAC	new byte[]{}	Collections.emptyMap()	CREATO CON SUCCESSO
4	3	2	2	DUMMY	null	Collections.emptyMap()	NullPointerException

Stesse considerazioni a priori non si possono fare, invece, sul metodo `createLedgerAdv`, se anche con gli stessi parametri di `createLedger`, il metodo offre delle maggiori funzionalità nella creazione di un ledger. Avendo stessi parametri di input, però, si utilizza il medesimo partizionamento ed analisi

dei valori ai bordi, per poi passarli con un approccio multidimensionale al test; analizzando l'output, si nota che il test per ogni entry restituisce il medesimo risultato analizzato per il metodo `public LedgerHandle createLedger(int ensSize, int writeQuorumSize, int ackQuorumSize, DigestType digestType, byte[] passwd, final Map<String, byte[]> customMetadata)`.

Rianalizzando la coverage della classe, si può notare un aumento, passando da un valore di statement-coverage del 38% al 49% e di branch-coverage dal 21% al 36%.

Prendendo in considerazione, invece, la classe *ReadCache*, per aumentare la coverage si procede al miglioramento del test del metodo `put`, essendo metodo all'interno della classe con coverage al di sotto del 100%. Si è proceduto con uno stress maggiore del metodo, testando non soltanto l'inserimento in cache di una sola entry ma bensì l'inserimento consecutivo di n entry; per fare ciò si è inserito un nuovo parametro booleano tra il set di input al test, che se impostato a `true` effettua l'inserimento consecutivo.

Rianalizzando la coverage si può notare come questa sia migliorata passando la statement-coverage dal 48% al 100% [***Figura 10***], e data coverage con coppie def-use coperte da 33 a 70.

PIT

Per completare la verifica della bontà e dell'adeguatezza dei test, differentemente dai criteri white-box e black-box precedenti, si procede tramite l'utilizzo del tool Pitest alla generazione e conseguente introduzione di alterazioni artificiali nel SUT sulle classi di interesse; quindi, portando mutazioni nel SUT e verificando che i test sviluppati e migliorati fino ad ora siano in grado di coprire le alterazioni introdotte. Dato il mutation score come il numero di mutanti rilevati, quindi "uccisi" (D), dall'insieme di test, rispetto al numero di mutanti rimasti non rilevati (L):

$$mutation\_score = \frac{|\mathrm{D}|}{|\mathrm{L}| + |D|}$$

dall'analisi [***Figura 11***] risulta come la classe BookKeeper abbia un [1]mutation score pari al 47%, e la classe ReadCache abbia un [2]mutation score pari al 68%.

$$^{1}mutation\_score = \frac{66}{140} = 0.471 \qquad ^{2}mutation\_score = \frac{41}{60} = 0.683$$

RELIABILITY

Dato che nel testing non ha senso parlare di correttezza, di seguito si procede con l'analisi dei profili operazionali e la conseguente stima della reliability; stima effettuata a livello di classe, considerando le combinazioni di parametri testate. Data PFD come la probabilità di failure rispetto alla richiesta, di seguito la formula della reliability:

$$reliability = 1 - PFD$$

La probabilità d'utilizzo di ogni combinazione di parametri presente all'interno del profilo operazionale, come da specifica, è stata assunta uniforme; quindi, una classe testata con k combinazioni di parametri, avrà una probabilità d'utilizzo con una determinata combinazione pari a $1/\mathrm{k}$. Il calcolo delle reliability quindi si è fatto partendo dal valore 1 e per ogni combinazione di parametri del profilo operazionale che fallisce, vi è stato sottratto $1/\mathrm{k}$.

Per la classe BookKeeper, sono stati eseguiti 74 test, di cui 11 hanno evidenziato fallimento, avendo così una reliability pari a 0.851; per la classe ReadCache, invece, sono stati eseguiti 27 test, di cui nessuno ha evidenziato fallimento, avendo così una reliability pari a 1.

SYNCOPE

Syncope è uno strumento che consente di centralizzare e gestire in modo efficiente le identità degli utenti, i loro permessi e l'accesso alle risorse all'interno delle organizzazioni.

Il criterio che ha portato la scelta delle classi a seguire è stato il loro livello di importanza ma soprattutto il loro livello di chiarezza e quindi la presenza di documentazione, anche se **minima** per l'intero progetto, che ci ha permesso di perseguire un approccio il più possibile black-box; le classi da testare scelte sono state:

`org.apache.syncope.core.spring.security.DefaultPasswordGenerator`

`org.apache.syncope.core.provisioning.api.utils.RealmUtils`

DeafaultPasswordGenerator.java

La classe DeafaultPasswordGenerator si occupa della generazione automatica di uno degli elementi chiave del sistema Syncope e cioè, come anche suggerito dal nome della classe, la *password*, seguendo determinate policies di creazione. Si procede con il testing del metodo principale della classe:

- `public String generate(final List<PasswordPolicy> policies)`

Il metodo **generate(..)** ha il compito di generare una password in accordo alla lista di policies passata, unico parametro di input. Al termine dell'esecuzione del metodo la password sarà ritornata in output in formato `String`.

Tramite l'utilizzo della documentazione [*Link 5*], si è proceduto con l'analisi del dominio dell'unico parametro di input, la policy. Quando si definisce una policy devono essere fornite le seguenti informazioni:

- *"allow null password"* – indica se la password è opzionale per gli utenti o meno, quindi se può essere rispettivamente nulla o non nulla;
- *"history length"* – quanti valori devono essere considerati;
- *"rules"* – insieme di regole, vincoli da applicare alla password, che definiscono la politica corrente. Dove le regole di default sono le seguenti:

    - "*maximum length* – lunghezza massima consentita (0 significa nessun limite impostato);
    - "*minimum length*" – lunghezza minima consentita (0 significa nessun limite impostato);
    - "*alphabetical*" – numero di caratteri alfabetici richiesti;
    - "*uppercase*" – numero di caratteri maiuscoli richiesti;
    - "*lowercase*" - numero di caratteri minuscoli richiesti;
    - "*digit*" – numero di caratteri numerici richiesti;
    - "*special*" – numero di caratteri speciali richiesti;
    - "*special chars*" – insieme di caratteri speciali consentiti;
    - "*illegal chars*" – insieme di caratteri non consentiti;
    - "*repeat same*" – dimensione massima che può avere una sequenza di caratteri ripetuta;
    - "*username allowed*" – se è possibile utilizzare un nome utente;
    - "*words not permitted*" – elenco di parole che non possono essere presenti.

    Queste possono essere estese tramite l'utilizzo delle Passay API [*Link 6*].

Per il partizionamento del dominio in input si procede seguendo la linea guida per un valore di tipo complesso:

- policies: null, {“valid_policies”}, {“invalid_policies”}

Date le criticità dei valori presenti ai bordi del dominio, si procede allora loro analisi, come segue:

- policies: null; {“empty”}; {“valid_policy”}; {“invalid_policy”};

Una policy non valida, si intende una policy che ha al suo interno almeno una rule invalida che porta una failure nel sistema. In questo caso si è provato a passare una policy contenente una rule invalida, dove la lunghezza della password richiesta è un numero negativo.

Per la realizzazione del seguente test si è fatto uso del framework Mockito per simulare parte dell’ambiente di esecuzione.

Essendo un unico parametro si è proseguito per il passaggio dei parametri in input al test, tramite un approccio unidimensionale; approccio, in questo caso, equivalente ad uno multidimensionale, essendo quest’ultimo combinazione lineare di tutte le classi di equivalenza su tutti i parametri.

#N.	policies	OUTPUT
0	`valid_policies`	CREATA CON SUCCESSO
1	`invalid_policies`	CREATA CON SUCCESSO - DEFAULT POLICY
2	`null`	NullPointerException
3	`new ArrayList<PasswordPolicy>()`	CREATA CON SUCCESSO - DEFAULT POLICY

Dai risultati dei test si è potuto notare come sia l’interfaccia implementata della policy `PasswordPolicy` sia il metodo di generazione delle password, presentino un ottimo livello di robustezza tale da non esser riuscito a generare un’istanza impattante non valida e che quindi generasse una failure, sia tramite il passaggio di policies contenenti rule non valide sia tramite, come effettuato anche nel progetto BookKeeper per il testing del metodo `createLedgers` della classe `BookKeeper`, la re-implementazione dell’interfaccia. In qualsiasi scenario non valido che si è provato ad implementare, questo viene sempre gestito restituendo una policy di default che impone sulla password una lunghezza minima di 8 caratteri e massima di 64 caratteri. Unico scenario in cui si è riuscito a stimolare il sistema verso un’eccezione è stato passando delle policies `null`.

RealmUtils.java

La classe RealmUtils ha il compito di fornire dei metodi per la gestione dei realm, oggetti che definiscono degli alberi gerarchici per i domini di sicurezza di utenti, gruppi e qualsiasi altro oggetto collegato. I metodi presi in considerazione nel testing della classe sono i seguenti:

- `public static String getGroupOwnerRealm(final String realmPath, final String groupKey)`
- `public static Optional<Pair<String, String>> parseGroupOwnerRealm(final String input)`
- `public static boolean normalizingAddTo(final Set<String> realms, final String newRealm)`
- `public static Pair<Set<String>, Set<String>> normalize(final Collection<String> realms)`

In accordo con la documentazione [*Link 7*], gli oggetti realm non sono altro che collezioni di stringhe dove ogni realm ha un parent ad eccezione del realm root che viene indentificato con ”/”, ed ogni realm è identificato univocamente dal suo path partendo dalla radice; ad esempio /a/b/c identifica il realm `c` appartenente al sotto-albero `b` avente a sua volta come realm genitore la radice `a`.

Il metodo **getGroupOwnerRealm(..)** genera in output una stringa identificativa per un `GroupOwner`, dal formato `path@key`, tramite il passaggio in input del realm e della chiave identificativa del gruppo. Per il metodo si è proceduto con il seguente partizionamento del dominio, seguendo le linee guida per i valori di tipo stringa:

- realmPath: null, {“valid_string_path”}, {“invalid_string_path”}

- groupKey: null, {“valid_string_key”}, {“invalid_string_key”}

Come scritto precedentemente, dalla documentazione, si può considerare un realmPath non valido, una stringa che non rispetti il paradigma “/a/b/c” come, ad esempio, la stringa “pr0v@”; ed eseguendo l’analisi con un approccio più white-box si può ricavare come una qualsiasi key per essere considerate valida debba seguire il seguente pattern regex:

```
[\p{L}\p{gc=Mn}\p{gc=Me}\p{gc=Mc}\p{Digit}\p{gc=Pc} \-@.~]+
```

Quindi si può considerare una stringa groupKey non valida una stringa che non rispetti il pattern, ad esempio la stringa “~~~”. Per le criticità dei valori ai bordi si esegue la seguente analisi dei boundary values:

- realmPath: null; “”; “/”; ”pr0v@”;

- groupKey: null; “”; “k3y”, “~~~”;

Per il passaggio dei parametri in input al test si è optato ad un approccio unidimensionale, data la semplicità, esposta precedentemente, del metodo.

#N.	realmPath	groupKey	OUTPUT
0	`null`	“”	“null@”
1	“”	“k3y”	“@k3y”
2	“/”	`null`	“/@null”
3	”pr0v@”	“~~~”	pr0v@@~~~

Dai risultati del test si evince come, indipendentemente dalla validità del path del realm e della key all’interno del sistema Syncope, non presentando all’interno del metodo nessun meccanismo di controllo e validazione dell’input ma solo una semplice concatenazione delle stringhe tramite l’operatore +, non si è riuscito a sollecitarlo in modo tale da scaturire una failure.

Il metodo **parseGroupOwnerRealm(..)**, in maniera opposta al metodo precedente, passata una stringa in input nel formato `path@key`, esegue il suo split sul carattere @ e ritorna in output un oggetto di tipo Pair con al suo interno le due stringhe splittate. Per il metodo si è proceduto con il seguente partizionamento del dominio, seguendo le linee guida per i valori di tipo stringa:

- input: null, {“valid_string_input”}, {“invalid_string_input”}

Per le criticità dei valori ai bordi si esegue la seguente analisi dei boundary values:

- input: null; “”; “path@key”; ”path-key”;

Per il passaggio dei parametri in input al test si è optato ad un approccio unidimensionale, come segue:

#N.	input	OUTPUT
0	`null`	`NullPointerException`
1	“”	`Optional.empty()`
2	“path@key”	`PARSE EFFETTUATA CON SUCCESSO`
3	”path-key”	`Optional.empty()`

Dai risultati del test si può notare che, come per il metodo precedentemente testato, indipendentemente dalla validità del path o della key, viene effettuato a priori lo split; se passata una stringa non valida, quindi che non rispetta il modello `path@key`, viene ritornato un oggetto `Optional.empty()`, altrimenti se stimolato tramite il passaggio di un input `null` viene scaturita l’eccezione `NullPointerException`.

Il metodo **normalizingAddTo(..)**, si occupa dell’inserimento di un nuovo realm in un insieme già esistente di realm, entrambi passati come parametri in input; restituisce in output `true` se l’inserimento è andato a buon fine altrimenti `false`. Per il metodo si è proceduto con il seguente partizionamento del dominio, seguendo le linee guida per i valori di tipo stringa:

- realms: null, {“valid_string”}, {“invalid_string”}

- newRealm: null, {“valid_string”}, {“invalid_string”}

Non essendo indicato nella documentazione quando un insieme di realm può essere considerato non valido all’interno del sistema, tramite un approccio white-box si cerca di analizzare il contenuto del metodo; dall’analisi si evince che per l’aggiunta di un realm nell’insieme si hanno tre scenari:

a) se tra tutti i realm presenti nell’insieme realms ce ne è uno che corrisponde per la sua interezza all’inizio di newRealm, quest’ultimo non viene aggiunto all’insieme, ritornando `false`;
b) se newRealm coincide per la sua interezza all’inizio di un realm che si trova all’interno dell’insieme, quest’ultimo viene rimosso dall’insieme ed al suo posto viene aggiunto newRealm, ritornando `true`;
c) se nessuno degli scenari precedenti, newRealm viene aggiunto direttamente nell’insieme realms, ritornando `true`.

Quindi dalle seguenti relazioni che lega newRealm ad un insieme realms, possiamo identificare meglio le classi di equivalenza, dicendo che realms e newRealm sono *validi* (c), quindi viene restituito `true` dal metodo e newRealm viene aggiunto, se quest’ultimo non inizia con nessun realm dell’insieme, e viceversa. Per quanto riguarda le classi di equivalenza {“invalid_string”} e {“valid_string”}, invece, vengono rimodellate, data la forte correlazione tra i parametri, non si può affermare a priori la validità di un newRealm senza analizzare l’insieme realms e viceversa; quindi, si sostituisce la classe di equivalenza {“valid_string”} con {“not_empty”} e si introducono le classi di equivalenza {“starts_with_newRealm”} (b) e {“starts_with_realms”} (a), come segue:

- realms: null, {“empty”}, {“not_empty”}

- newRealm: null, {“empty”}, {“not_empty”}, {“starts_with_realms”}, {“starts_with_newRealm”}

Per le criticità dei valori ai bordi si esegue la seguente analisi dei boundary values:

- realms: null; []; ["/a/b",”/c/d”];

- newRealm: null; “”; “/e/f”; “/a/b/c”; “/a”;

Per il passaggio dei parametri in input al test si è optato ad un approccio multidimensionale, come segue, dove la colonna “realms after exec.” rappresenta il contenuto dell’insieme realms dopo l’esecuzione del metodo.

#N.	realms	newRealm	realms after exec.	OUTPUT
0	`null`	`null`		`NullPointerException`
1	`null`	“”		`NullPointerException`
2	`null`	“/e/f”		`NullPointerException`
3	`null`	“/a/b/c”		`NullPointerException`
4	`null`	“/a”		`NullPointerException`
5	`[]`	`null`	`[null]`	`true`
6	`[]`	“”	[“”]	`true`
7	`[]`	“/e/f”	[“/e/f”]	`true`
8	`[]`	“/a/b/c”	[“/a/b/c”]	`true`
9	`[]`	“/a”	[“/a”]	`true`
10	`["/a/b",”/c/d”]`	`null`		`NullPointerException`
11	`["/a/b",”/c/d”]`	“”	[“”]	`true`
12	`["/a/b",”/c/d”]`	“/e/f”	`["/a/b",”/e/f”,”/c/d”]`	`true`
13	`["/a/b",”/c/d”]`	“/a/b/c”	`["/a/b",”/c/d”]`	`false`
14	`["/a/b",”/c/d”]`	“/a”	`["/a",”/c/d”]`	`true`

Dai risultati dei test è emerso come se passato almeno uno dei parametri `null`, verrà scaturita un’eccezione del tipo `NullPointerException`; inoltre, si può notare come le relazioni precedenti (a) (b) (c) vengono rispettate tranne nei casi in cui il parametro newRealm è vuoto, generando un’anomalia, dato che se newRealm è vuoto non dovrebbe modificare il contenuto dell’insieme realms (configurazione 11).

Dato il comportamento anomalo tramite una stringa newRealm vuota, si è continuato il testing del metodo tramite l’implementazione di un nuovo set di input in cui nell’insieme realms è presente solo un elemento realm vuoto:

15	[“”]	`null`		`NullPointerException`
16	[“”]	“”	[“”]	`false`
17	[“”]	“/e/f”	[“”]	`false`
18	[“”]	“/a/b/c”	[“”]	`false`
19	[“”]	“/a”	[“”]	`false`

Analizzando i risultati dei test con quest’ultimo set di input si può notare come l’anomalia si presenti anche nei casi in cui nell’insieme realms ci sia un realm vuoto; infatti, se newRealm non è vuoto dovrebbe essere aggiunto all’interno dell’insieme realms.

Analizzando il codice del metodo possiamo attribuire queste **failure** ad un possibile **BUG** nel codice, dato che quando viene passato un parametro stringa vuota al metodo `string.startsWith(string)`, per la sua implementazione nativa restituirà il valore `true` [*Link 8*]; quindi, per le configurazioni 17, 18 e 19 non facendo entrare nella condizione per l’aggiunta di newRealm, e per la configurazione 11 entrando nella condizione di rimozione per ogni elemento dell’insieme realms [*Figura 12*].

Si vuole sottolineare la gravità del presunto BUG in quanto, tramite il passaggio di una semplice stringa vuota “”, si va a cancellare l’interno contenuto della struttura realms.

Il metodo **normalize(..)**, ha il compito di normalizzare una collezione di realm passata come parametro in input, dividendo il suo contenuto in due set di stringhe da ritornare come output, uno contenente tutti i realm non riferiti da un gruppo e l'altro contenente tutti i realm della forma `path@groupOwnerKey`. Essendo l'unico parametro un set di stringhe, si è proceduto con il seguente partizionamento del dominio:

- realms: null, {"valid_string"}, {"invalid_string"}

Un insieme realms si considera {"valid_string"} se presenta almeno un realm della forma `path@groupOwnerKey` ed {"invalid_string"} se non presenta nessun realm della forma `path@groupOwnerKey`. Per le criticità dei valori ai bordi si esegue la seguente analisi dei boundary values:

- realms: null; []; ["/a/b@k3y"]; ["/c/d"];

Tramite un approccio unidimensionale si sono passati i seguenti parametri di input ai test:

#N.	realms	OUTPUT
0	`null`	`([],[])`
1	`[]`	`([],[])`
2	`["/a/b@k3y"]`	`([],["/a/b@k3y"])`
3	`["/c/d"]`	`(["/c/d"],[])`

Dai risultati dei test si può notare come tutti i test rispettino i comportamenti attesi ed in particolar modo come non si sia riusciti a generare failure.

ADEGUATEZZA DEI TEST e MIGLIORAMENTO

Di seguito si andranno ad analizzare tramite due diversi criteri di copertura, l'adeguatezza e la bontà dei test sviluppati, entrambi basati sul cotrol-flow graph che, quindi, descrivono le relazioni di dipendenza tra le varie istruzioni del SUT, tramite l'utilizzo del tool JaCoCo. Differentemente da quanto analizzato nel progetto BookKeeper, per il progetto Syncope non si è potuto analizzare i test sviluppati con un criterio di copertura basato sul data-flow graph che descrive l'uso di variabili/dati all'interno del SUT, a causa dell'incompatibilità del tool Ba-Dua che per il suo corretto funzionamento richiede la JDK 8, con la JDK minima richiesta da Syncope, versione 17.

JACOCO

La classe *DefaultPasswordGenerator* presenta uno statement-coverage del 60% ed un branch-coverage del 45%; in particolare, si ha un livello di coverage elevato [*Figura 13*] grazie al metodo testato che internamente invoca quasi tutti gli altri metodi della classe.

Analizzando il report, si nota come il metodo `protected DefaultPasswordRuleConf merge(final List<DefaultPasswordRuleConf> defaultRuleConfs)` che si occupa della validazione delle policy assegnate per la creazione della password, abbia un numero elevato di branch inesplorati [*Figura 14*]; si procede quindi, con la stimolazione del metodo, operando sullo stato dell'istanza della classe `DefaultPasswordRuleConf`, la quale contiene i settaggi di tutte le rules caratterizzanti una determinata policy. Di seguito, si procede con il partizionamento del dominio per ogni rule assegnabile ad una policy:

- maxLength: {<=0},{>0}
- minLength: {<=0},{>0}
- alphabetical: {<=0},{>0}

- uppercase: {<=0},{>0}
- lowercase: {<=0},{>0}
- digit: {<=0},{>0}
- special: {<=0},{>0}
- specialChars: {“empty”},{“not_empty”}
- illegalChars: {“empty”},{“not_empty”}
- repeatSame: {<=0},{>0}
- usernameAllowed: {true},{false}
- wordsNotPermitted: {“empty”},{“not_empty”}

Non permettendo il settaggio di una regola nulla, nel partizionamento del dominio non si è preso in considerazione il valore `null`, inoltre, aumentando le configurazioni e quindi aumentando lo spettro di rule per le password generate, si è implementato un nuovo algoritmo per il controllo della correttezza della password generata, basato sul matching di pattern regex appositi. Dai risultati dei test con le nuove configurazioni si può notare, come notato anche nell’analisi iniziale del metodo, che per valori negativi o settati a 0 questi vengano gestiti con delle regole di default; oltretutto si è riusciti a sollecitare il sistema passando 1 come dimensione massima per una sequenza di caratteri ripetuti scaturendo l’eccezione `java.lang.IllegalArgumentException`, ed impostando una larghezza massima minore, rispetto a quanto dovrebbe essere per rispettare tutte le ulteriori rule settate, scaturendo l’eccezione `java.nio.BufferOverflowException`.

Per aumentare ulteriormente la coverage si è proceduto con il testing del metodo `public String generate(final ExternalResource resource, final List<Realm> realms)` che a differenza della variante precedentemente testata, genera una password seguendo le policies appartenenti all’istanza della classe che implementa ExternalResource od ai realm, passati entrambi come parametri in input. Non si procede con il partizionamento del dominio di input dato che resource e realms sono due implementazioni delle corrispettive interfacce, modellato in modo da immagazzinare e restituire una PasswordPolicy, e quindi un dominio già trattato ed analizzato nel test precedenti come il set dei valori da testare, proprio per questo il suo test è stato aggiunto nella classe di testing per il metodo `public String generate(final List<PasswordPolicy> policies)` passando i medesimi set di input, ritornando in output i medesimi risultati.

Rianalizzando la coverage della classe, grazie a questi miglioramenti, si può notare un notevole aumento della coverage, unico branch che non si è riuscito ad esplorare è stato quello all’interno del metodo `protected List<PasswordRule> getPasswordRules(final PasswordPolicy policy)` dove non si è riuscito a modellare uno scenario in grado di scaturire un eccezione [*Figura 15*]. Analizzando l’aumento si passa da un valore di statement-coverage del 60% al 96% e da un valore di branch-coverage del 45% al 86% [*Figura 16*].

La classe *RealmUtils* presenta uno statement-coverage del 60% ed un branch-coverage del 77%, in particolare:

il metodo `public static String getGroupOwnerRealm(final String realmPath, final String groupKey)` è caratterizzato da uno statement-coverage del 100% e di una branch-coverage indefinita;

il metodo `public public static Optional<Pair<String, String>> parseGroupOwnerRealm (final String input)` è caratterizzato da uno statement-coverage del 100% e di una branch-coverage del 75%. Analizzando il report si nota come non sia stato esplorato la branch di riga 38 dove `split==null` [*Figura 17*], analizzando la documentazione del metodo `string.split()` si può leggere come questo non ritorni mai null [*Link 9*], quindi non raggiungibile.

il metodo `public static Pair<Set<String>, Set<String>> normalize(final Collection <String> realms)` è caratterizzato da uno statement-coverage del 100% e di una branch-coverage del 100%;

il metodo `public void put(long ledgerId, long entryId, ByteBuf entry)` è caratterizzato da uno statement-coverage del 100% e di una branch-coverage del 100%.

Per migliorare la coverage della classe si procede con il testing del metodo `public static Set<String> getEffective(final Set<String> allowedRealms, final String requestedRealm)`, incaricato nel ritornare in output un insieme di realm filtrato e normalizzato, dati in input due set di realm ed un singolo realm; a differenza di tutti gli altri metodi testati, in mancanza di documentazione e commenti, per la sua comprensione è stato usato un approccio maggiormente white-box. Come analizzato in precedenza un set di realm si considera invalido se presenta almeno un realm invalido, e quindi un realm che possiede un path che non rispetta il modello "/a/b/c" e valido se non possiede realm invalidi; di seguito si procede con il partizionamento del dominio di input nelle varie classi di equivalenza:

- allowedRealms: null, {"valid"}, {"invalid"}

- requestedRealm: null, {"valid"}, {"invalid"}

Per le criticità dei valori ai bordi si esegue la seguente analisi dei boundary values:

- allowedRealms: null; []; ["/a/b/c@key1"]; ["a-b-c@key1"];

- requestedRealm: null; ""; "/e/f/g@key2"; "a-b-c@key2";

Tramite un approccio unidimensionale si sono passati i seguenti parametri di input ai test:

#N.	allowedRealms	requestedRealm	OUTPUT
0	`["/a/b/c@key1"]`	`"/e/f/g@key2"`	`["/a/b/c@key1"]`
1	`null`	`"/e/f/g@key2"`	`[]`
2	`["/a/b/c@key1"]`	`null`	`NullPointerException`
3	`[]`	`"/e/f/g@key2"`	`[]`
4	`["/a/b/c@key1"]`	`""`	`["/a/b/c@key1"]`
5	`["a-b-c@key1"]`	`"/e/f/g@key2"`	`["a-b-c@key1"]`
6	`["/a/b/c@key1"]`	`"a-b-c@key2"`	`["/a/b/c@key1"]`

Dai risultati dei test si evince come, riscontrato anche nei test precedenti, non avviene nessun controllo sulla validità sintattica dei realm ed inoltre, a differenza di allowedRealms, non venie gestito dal metodo lo scenario in cui requestRealm sia nullo così da scaturire un'eccezione del tipo `NullPointerException`.

Rianalizzando la coverage della classe, grazie ai miglioramenti apportati, si passa da un livello di statement-coverage del 60% al 100% e da un livello di branch-coverage del 77% al 95%.

PIT

Per completare la verifica della bontà e dell'adeguatezza dei test, differentemente dai criteri white-box e black-box precedenti, si procede tramite l'utilizzo del tool Pitest alla generazione e conseguente introduzione di alterazioni artificiali nel SUT sulle classi di interesse; quindi, portando mutazioni nel SUT e verificando che i test sviluppati e migliorati fino ad ora siano in grado di coprire le alterazioni introdotte. Dato il mutation score come il numero di mutanti rilevati, quindi "uccisi" (D), dall'insieme di test, rispetto al numero di mutanti rimasti non rilevati (L):

$$mutation\_score = \frac{|\mathrm{D}|}{|\mathrm{L}| + |D|}$$

dall'analisi risulta come la classe *DefaultPasswordGenerator* [*Figura 18*] abbia una [1]mutation score pari al 36%, e la classe *RealmUtils* [*Figura 19*] abbia una [2]mutation score pari al 78%, sottolineando una certa robustezza dei test alle mutazioni.

$$^{1}mutation\_score = \frac{23}{65} = 0.356 \qquad ^{2}mutation\_score = \frac{14}{18} = 0.777$$

ANALISI DELLE CLASSI e TEST DI INTEGRAZIONE

I test di integrazione hanno l'obiettivo di far emergere malfunzionamenti derivanti dall'interazione tra due o più componenti software, dove a differenza del test di unità le classi si prendono singolarmente in considerazione già testate e perfettamente funzionanti.

Analizzando le classi precedentemente testate, l'unica classe che interagisce con un'altra di cui è presente documentazione e commenti, anche se entrambi minimi, e che quindi si possano effettuare test di unità e testare il suo comportamento di integrazione, è RealmUtils; in particolare, il metodo `public static Pair<Set<String>, Set<String>> normalize(final Collection<String> realms)` che interagisce con il metodo `protected Set<SyncopeGrantedAuthority> buildAuthorities(final Map<String, Set<String>> entForRealms)` della classe AuthDataAccessor. Prima di procedere al test di integrazione si procede con il test di unità per la classe AuthDataAccessor.

La classe AuthDataAccessor ha il compito di gestire le autenticazioni degli utenti e le loro autorizzazioni verso le risorse del sistema. Dalla documentazione [*Link 10*] si possono identificare gli elementi cardine della classe che sono i *realm*, già precedentemente trattati, gli *entitlement* che non sono altro che stringhe, staticamente definite [*Link 11*], che descrivono il diritto di eseguire un'operazione su Syncope e i *SyncopeGrantedAuthority*, oggetti utilizzati per rappresentare l'autorizzazione (o l'authority) di un utente o di un ruolo all'interno del sistema.

Il metodo **buildAuthorities(..)**, dato un oggetto `Map<String, Set<String>> entForRealms` contenente per ogni set di path realm un entitlement, si occupa di normalizzare l'insieme dei realm e generare con il successivo ritorno in output il set delle relative autorizzazioni, quindi il set di oggetti SyncopeGrantedAuthority associato. Di seguito si procede con il partizionamento del dominio di input nelle relative classi di equivalenza, seguendo le linee guida per un dato di tipo complesso:

- entForRealms: null, {"valid_instance"}, {"ïnvalid_instance"}

Data la criticità dei valori ai bordi assunti dalle classi di equivalenza si procede di seguito con l'analisi dei boundary values:

- entForRealms: null; {"empty"}; {"valid"}; {"invalid_realm"}; {"invalid_entitlement"};

Non essendo espressamente specificato nella documentazione quando un entForRealms fosse invalido, lo si considera appartenente alla classe {"ïnvalid_instance"} quando possiede una coppia che al suo interno ha almeno un entitlement invalido, quindi che non è definito all'interno della lista dei valori assumibili e.g. "`fake_ENT_ACCESS`", oppure ha un set di path realm invalido, quindi che possiede almeno un path realm che non rispetta la forma "/a/b/c" e.g. "`fake-path-realm`". Di seguito i valori passati al test:

#N.	entForRealms	OUTPUT
0	`null`	`NullPointerException`
1	`Collections.emptyMap()`	`CREAZIONE SET VUOTO`
2	`<"DOMAIN_READ", ["/a/b/c"]>`	`SET CREATO CON SUCCESSO`
3	`<"fake_ENT_ACCESS", ["/a/b/c"]>`	`SET CREATO CON SUCCESSO`
4	`<" DOMAIN_READ", ["fake-path-realm"]>`	`SET CREATO CON SUCCESSO`

Dai risultati dei test si può notare come indipendentemente dal contenuto dell'oggetto entForRealms viene creato il corrispettivo set di autorizzazioni, e l'unico scenario che scaturisce un'eccezione è quello in cui viene passato il parametro `null`. Si è provato tramite la reimplementazione dell'interfaccia ed oltretutto tramite l'utilizzo di Mockito, di istanziare un oggetto entForRealms non valido che scaturisse un'eccezione, senza però avere successo, riuscendo solo a scaturire la generazione di un set di autorizzazioni vuoto.

Per valutare l'adeguatezza del test sviluppato si analizza tramite il tool di control-flow coverage JaCoCo, da cui possiamo notare dal report generato [*Figura 20*] come si sia raggiunto il massimo livello di copertura.

TEST DI INTEGRAZIONE

Dopo aver svolto i test d'unità documentati nelle sezioni precedenti, si procede con la realizzazione di test di integrazione tra le classi RealmUtils e AuthDataAccessor, rispettivamente tra i loro metodi `public static Pair<Set<String>, Set<String>> normalize(final Collection<String> realms)` e `protected Set<SyncopeGrantedAuthority> buildAuthorities(final Map<String, Set<String>> entForRealms)`; tramite l'utilizzo di una strategia *big-bang*.

Per generare un'autorizzazione vengono passati un insieme di entitlement e realms al metodo `buildAuthorities`, questo durante la fase di generazione del set di autorizzazioni, oggetti di tipo SyncopeGrantedAuthority, compirà la normalizzazione dell'insieme dei realms passati, una per ogni set generato, tramite il metodo `normalize` della classe RealmUtils, precedentemente analizzata.

Nei test di integrazione sviluppati, inizialmente è stata testata la raggiungibilità tra le due classi, tramite la realizzazione del metodo di test `normalizeCallTest()`, che tramite l'utilizzo di Mockito ed in particolar modo del suo metodo *verify*, verifica che all'invocazione del metodo `buildAuthorities` venga invocato una ed una sola volta il metodo `normalize`; obiettivo raggiunto tramite un previo mocking della classe RealmUtils e dell'utilizzo del metodo *when* delle Mockito API.

Successivamente si è sviluppato il metodo di test `normalizeWorkingTest()`, che verifica il corretto funzionamento del metodo `normalize` all'interno del metodo `buildAuthorities` invocato; tramite diversi set di input, dopo l'invocazione del metodo `buildAuthorities`, si preleva l'insieme dei realms normalizzati all'interno dell'autorizzazione generata in output, e si confronta l'insieme con quello generato da una normalizzazione diretta effettuata nel metodo, verificando l'uguaglianza tra i due insieme. Dai risultati del test si nota come, dati gli stessi insieme non normalizzati di realms, vengono restituiti gli stessi insiemi normalizzati, e nel caso di un insieme di realms nullo si ha una propagazione attesa dell'eccezione `java.lang.NullPointerException`.

RELIABILITY

Dato che nel testing non ha senso parlare di correttezza, di seguito si procede con l'analisi dei profili operazionali e la conseguente stima della reliability; stima effettuata a livello di classe, considerando le combinazioni di parametri testate.

Per la classe DeafaultPasswordGenerator, sono stati eseguiti 33 test, di cui nessuno ha evidenziato fallimento, avendo così una reliability pari a 1; per la classe RealmUtils, invece, sono stati eseguiti 45 test, di cui 4 hanno evidenziato fallimento, avendo così una reliability pari a 0.91.

Per la classe AuthDataAccessor, sono stati eseguiti 16 test, di cui nessuno ha evidenziato fallimento, avendo così una reliability pari a 1.

CONCLUSIONI FINALI

Anche se il progetto è stato discretamente complesso, complessità dovuta a progetti come Syncope: progetto in mutamento continuo/giornaliero in cui non sono presenti commenti nel codice e non è presente una documentazione sufficientemente esaustiva per il testing; si ritiene che i risultati ottenuti siano stati soddisfacenti in quanto i tool di coverage utilizzati confermano, anche se migliorabile, una certa robustezza dei test sviluppati, i quali hanno permesso di individuare diversi malfunzionamenti all'interno dei due progetti.

RIFERIMENTI

Di seguito i riferimenti presenti nel documento

LINK

- **[Link 1] The BookKeeper Protocol -**
  https://bookkeeper.apache.org/docs/next/development/protocol#ledger-metadata
- **[Link 2] BokKeeper Configuration –**
  https://bookkeeper.apache.org/docs/next/reference/config
- **[Link 3] From Java code to Java heap -**
  https://developer.ibm.com/articles/j-codetoheap/
- **[Link 4] ByteBuf (Netty API Reference) -**
  https://netty.io/4.1/api/io/netty/buffer/ByteBuf.html
- **[Link 5] Syncope Password -**
  https://syncope.apache.org/docs/reference-guide.html#policies-password
- **[Link 6] Passay API Reference –**
  https://www.passay.org/reference/
- **[Link 7] Syncope Realms -**
  https://syncope.apache.org/docs/reference-guide.html#realms
- **[Link 8] Class String #startsWith -**
  https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html#startsWith%28java.lang.String%29
- **[Link 9] Class String #split -**
  https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#split-java.lang.String-
- **[Link 10] Syncope Entitlements -**
  https://syncope.apache.org/docs/reference-guide.html#entitlements
- **[Link 11] List of all static Entitlements –**
  https://github.com/apache/syncope/blob/syncope-3.0.4/common/idrepo/lib/src/main/java/org/apache/syncope/common/lib/types/IdRepoEntitlement.java

FIGURE

```
949.    public LedgerHandle createLedger(int ensSize, int writeQuorumSize, int ackQuorumSize,
950.                                     DigestType digestType, byte[] passwd, final Map<String, byte[]> customMetadata)
951.            throws InterruptedException, BKException {
952.        CompletableFuture<LedgerHandle> future = new CompletableFuture<>();
953.        SyncCreateCallback result = new SyncCreateCallback(future);
954.
955.        /*
956.         * Calls asynchronous version
957.         */
958.        asyncCreateLedger(ensSize, writeQuorumSize, ackQuorumSize, digestType, passwd,
959.                          result, null, customMetadata);
960.
961.        LedgerHandle lh = SyncCallbackUtils.waitForResult(future);
962.        if (lh == null) {
963.            LOG.error("Unexpected condition : no ledger handle returned for a success ledger creation");
964.            throw BKException.create(BKException.Code.UnexpectedConditionException);
965.        }
966.        return lh;
967.    }
```

Figura 1 – BookKeeper.java {createLedger()} analisi con JaCoCo

```
1361.     public void deleteLedger(long lId) throws InterruptedException, BKException {
1362.         CompletableFuture<Void> future = new CompletableFuture<>();
1363.         SyncDeleteCallback result = new SyncDeleteCallback(future);
1364.         // Call asynchronous version
1365.         asyncDeleteLedger(lId, result, null);
1366.
1367.         SyncCallbackUtils.waitForResult(future);
1368.     }
```

Figura 2 – BookKeeper.java {deleteLedger()} analisi con JaCoCo

```
1287.     public LedgerHandle openLedger(long lId, DigestType digestType, byte[] passwd)
1288.             throws BKException, InterruptedException {
1289.         CompletableFuture<LedgerHandle> future = new CompletableFuture<>();
1290.         SyncOpenCallback result = new SyncOpenCallback(future);
1291.
1292.         /*
1293.          * Calls async open ledger
1294.          */
1295.         asyncOpenLedger(lId, digestType, passwd, result, null);
1296.
1297.         return SyncCallbackUtils.waitForResult(future);
1298.     }
```

Figura 3 – BookKeeper.java {openLedger()} analisi con JaCoCo

```
92.     public void put(long ledgerId, long entryId, ByteBuf entry) {
93.         int entrySize = entry.readableBytes();
94.         int alignedSize = align64(entrySize);
95.
96.         lock.readLock().lock();
97.
98.         try {
99.             if (entrySize > segmentSize) {
100.                 log.warn("entrySize {} > segmentSize {}, skip update read cache!", entrySize, segmentSize);
101.                 return;
102.             }
103.             int offset = currentSegmentOffset.getAndAdd(alignedSize);
104.             if (offset + entrySize > segmentSize) {
105.                 // Roll-over the segment (outside the read-lock)
106.             } else {
107.                 // Copy entry into read cache segment
108.                 cacheSegments.get(currentSegmentIdx).setBytes(offset, entry, entry.readerIndex(),
109.                         entry.readableBytes());
110.                 cacheIndexes.get(currentSegmentIdx).put(ledgerId, entryId, offset, entrySize);
111.                 return;
112.             }
113.         } finally {
114.             lock.readLock().unlock();
115.         }
116.
117.         // We could not insert in segment, we to get the write lock and roll-over to
118.         // next segment
119.         lock.writeLock().lock();
120.
121.         try {
122.             int offset = currentSegmentOffset.getAndAdd(entrySize);
123.             if (offset + entrySize > segmentSize) {
124.                 // Rollover to next segment
125.                 currentSegmentIdx = (currentSegmentIdx + 1) % cacheSegments.size();
126.                 currentSegmentOffset.set(alignedSize);
127.                 cacheIndexes.get(currentSegmentIdx).clear();
128.                 offset = 0;
129.             }
130.
131.             // Copy entry into read cache segment
132.             cacheSegments.get(currentSegmentIdx).setBytes(offset, entry, entry.readerIndex(), entry.readableBytes());
133.             cacheIndexes.get(currentSegmentIdx).put(ledgerId, entryId, offset, entrySize);
134.         } finally {
135.             lock.writeLock().unlock();
136.         }
137.     }
```

Figura 4 – ReadCache.java {put()} analisi con JaCoCo

```
139.    public ByteBuf get(long ledgerId, long entryId) {
140.        lock.readLock().lock();
141.
142.        try {
143.            // We need to check all the segments, starting from the current one and looking
144.            // backward to minimize the
145.            // checks for recently inserted entries
146.            int size = cacheSegments.size();
147.            for (int i = 0; i < size; i++) {
148.                int segmentIdx = (currentSegmentIdx + (size - i)) % size;
149.
150.                LongPair res = cacheIndexes.get(segmentIdx).get(ledgerId, entryId);
151.                if (res != null) {
152.                    int entryOffset = (int) res.first;
153.                    int entryLen = (int) res.second;
154.
155.                    ByteBuf entry = allocator.buffer(entryLen, entryLen);
156.                    entry.writeBytes(cacheSegments.get(segmentIdx), entryOffset, entryLen);
157.                    return entry;
158.                }
159.            }
160.        } finally {
161.            lock.readLock().unlock();
162.        }
163.
164.        // Entry not found in any segment
165.        return null;
166.    }
167.
168.    public boolean hasEntry(long ledgerId, long entryId) {
169.        lock.readLock().lock();
170.
171.        try {
172.            int size = cacheSegments.size();
173.            for (int i = 0; i < size; i++) {
174.                int segmentIdx = (currentSegmentIdx + (size - i)) % size;
175.
176.                LongPair res = cacheIndexes.get(segmentIdx).get(ledgerId, entryId);
177.                if (res != null) {
178.                    return true;
179.                }
180.            }
181.        } finally {
182.            lock.readLock().unlock();
183.        }
184.
185.        // Entry not found in any segment
186.        return false;
187.    }
```

Figura 5 – ReadCache.java {get()} analisi con JaCoCo

```
168.    public boolean hasEntry(long ledgerId, long entryId) {
169.        lock.readLock().lock();
170.
171.        try {
172.            int size = cacheSegments.size();
173.            for (int i = 0; i < size; i++) {
174.                int segmentIdx = (currentSegmentIdx + (size - i)) % size;
175.
176.                LongPair res = cacheIndexes.get(segmentIdx).get(ledgerId, entryId);
177.                if (res != null) {
178.                    return true;
179.                }
180.            }
181.        } finally {
182.            lock.readLock().unlock();
183.        }
184.
185.        // Entry not found in any segment
186.        return false;
187.    }
```

Figura 6 - ReadCache.java {hasEntry()} analisi con JaCoCo

```
<method name="createLedger" desc="(IIILorg/apache/bookkeeper/client/BookKeeper$DigestType;[BLjava/util/Map;)Lorg/apache/bookkeeper/client/LedgerHandle;">
  <du var="LOG" def="952" use="963" covered="0"/>
  <du var="lh" def="961" use="962" target="963" covered="0"/>
  <du var="lh" def="961" use="962" target="966" covered="1"/>
  <du var="lh" def="961" use="966" covered="1"/>
  <counter type="DU" missed="2" covered="2"/>
  <counter type="METHOD" missed="0" covered="1"/>
</method>
```

Figura 7 – BookKeeper.java {createLedger()} analisi con BaDua

Figura 8 – ReadCache.java {put()} analisi con BaDua

```java
/**
 * Creates a new ledger. Default of 3 servers, and quorum of 2 servers.
 *
 * @param digestType
 *          digest type, either MAC or CRC32
 * @param passwd
 *          password
 * @return a handle to the newly created ledger
 * @throws InterruptedException
 * @throws BKException
 */
public LedgerHandle createLedger(DigestType digestType, byte[] passwd)
        throws BKException, InterruptedException {
    return createLedger(3, 2, digestType, passwd);
}
```

Figura 9 – BookKeeper.java {createLedger()}

```
92.     public void put(long ledgerId, long entryId, ByteBuf entry) {
93.         int entrySize = entry.readableBytes();
94.         int alignedSize = align64(entrySize);
95.
96.         lock.readLock().lock();
97.
98.         try {
99.             if (entrySize > segmentSize) {
100.                 log.warn("entrySize {} > segmentSize {}, skip update read cache!", entrySize, segmentSize);
101.                 return;
102.             }
103.             int offset = currentSegmentOffset.getAndAdd(alignedSize);
104.             if (offset + entrySize > segmentSize) {
105.                 // Roll-over the segment (outside the read-lock)
106.             } else {
107.                 // Copy entry into read cache segment
108.                 cacheSegments.get(currentSegmentIdx).setBytes(offset, entry, entry.readerIndex(),
109.                         entry.readableBytes());
110.                 cacheIndexes.get(currentSegmentIdx).put(ledgerId, entryId, offset, entrySize);
111.                 return;
112.             }
113.         } finally {
114.             lock.readLock().unlock();
115.         }
116.
117.         // We could not insert in segment, we to get the write lock and roll-over to
118.         // next segment
119.         lock.writeLock().lock();
120.
121.         try {
122.             int offset = currentSegmentOffset.getAndAdd(entrySize);
123.             if (offset + entrySize > segmentSize) {
124.                 // Rollover to next segment
125.                 currentSegmentIdx = (currentSegmentIdx + 1) % cacheSegments.size();
126.                 currentSegmentOffset.set(alignedSize);
127.                 cacheIndexes.get(currentSegmentIdx).clear();
128.                 offset = 0;
129.             }
130.
131.             // Copy entry into read cache segment
132.             cacheSegments.get(currentSegmentIdx).setBytes(offset, entry, entry.readerIndex(), entry.readableBytes());
133.             cacheIndexes.get(currentSegmentIdx).put(ledgerId, entryId, offset, entrySize);
134.         } finally {
135.             lock.writeLock().unlock();
136.         }
137.     }
```

Figura 10 - ReadCache.java {put()} analisi con JaCoCo

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
2	57%	244/431	54%	107/200

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.bookkeeper.bookie.storage.ldb	1	98%	86/88	68%	41/60
org.apache.bookkeeper.client	1	46%	158/343	47%	66/140

Report generated by PIT 1.5.1

Figura 11 – Report Pit BookKeeper

```java
boolean dontAdd = false;
        Set<String> toRemove = new HashSet<>();
        for (String realm : realms) {
            if (newRealm.startsWith(realm)) {
                dontAdd = true;
            } else if (realm.startsWith(newRealm)) {
                toRemove.add(realm);
            }
        }

        realms.removeAll(toRemove);
        if (!dontAdd) {
            realms.add(newRealm);
        }
        return !dontAdd;
```

Figura 12 - RealmUtils.java {normalizingAddTo()}

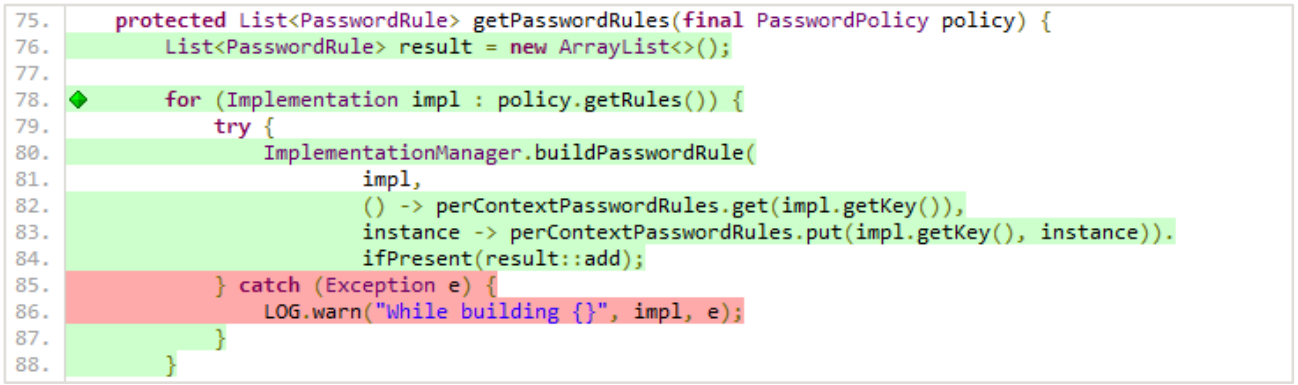
DefaultPasswordGenerator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
lambda$merge$10(DefaultPasswordRuleConf, DefaultPasswordRuleConf)		46%		50%	12	14	13	28	0	1
generate(ExternalResource, List)		0%		n/a	1	1	4	4	1	1
lambda$generate$1(List, Realm)		0%		n/a	1	1	3	3	1	1
lambda$merge$9(DefaultPasswordRuleConf, String)		0%		0%	2	2	1	1	1	1
lambda$merge$8(DefaultPasswordRuleConf, Character)		0%		0%	2	2	1	1	1	1
lambda$merge$7(DefaultPasswordRuleConf, Character)		0%		0%	2	2	1	1	1	1
lambda$generate$0(List, PasswordPolicy)		0%		0%	2	2	1	1	1	1
merge(List)		81%		50%	3	4	1	10	0	1
getPasswordRules(PasswordPolicy)		84%		100%	0	2	2	9	0	1
generate(DefaultPasswordRuleConf)		100%		100%	0	2	0	7	0	1
generate(List)		100%		n/a	0	1	0	3	0	1
lambda$generate$6(List, PasswordPolicy)		100%		n/a	0	1	0	3	0	1
DefaultPasswordGenerator()		100%		n/a	0	1	0	2	0	1
lambda$getPasswordRules$3(Implementation, PasswordRule)		100%		n/a	0	1	0	1	0	1
lambda$generate$5(List, PasswordRule)		100%		n/a	0	1	0	1	0	1
lambda$getPasswordRules$2(Implementation)		100%		n/a	0	1	0	1	0	1
lambda$generate$4(PasswordRule)		100%		n/a	0	1	0	1	0	1
static {...}		100%		n/a	0	1	0	1	0	1
Total	154 of 389	60%	24 of 44	45%	25	40	22	70	6	18

Figura 13 – DefaultPasswordGenerator.java analisi con JaCoCo

```
104.    protected DefaultPasswordRuleConf merge(final List<DefaultPasswordRuleConf> defaultRuleConfs) {
105.        DefaultPasswordRuleConf result = new DefaultPasswordRuleConf();
106.        result.setMinLength(VERY_MIN_LENGTH);
107.        result.setMaxLength(VERY_MAX_LENGTH);
108.
109.        defaultRuleConfs.forEach(ruleConf -> {
110.            if (ruleConf.getMinLength() > result.getMinLength()) {
111.                result.setMinLength(ruleConf.getMinLength());
112.            }
113.
114.            if (ruleConf.getMaxLength() > 0 && ruleConf.getMaxLength() < result.getMaxLength()) {
115.                result.setMaxLength(ruleConf.getMaxLength());
116.            }
117.
118.            if (ruleConf.getAlphabetical() > result.getAlphabetical()) {
119.                result.setAlphabetical(ruleConf.getAlphabetical());
120.            }
121.
122.            if (ruleConf.getUppercase() > result.getUppercase()) {
123.                result.setUppercase(ruleConf.getUppercase());
124.            }
125.
126.            if (ruleConf.getLowercase() > result.getLowercase()) {
127.                result.setLowercase(ruleConf.getLowercase());
128.            }
129.
130.            if (ruleConf.getDigit() > result.getDigit()) {
131.                result.setDigit(ruleConf.getDigit());
132.            }
133.
134.            if (ruleConf.getSpecial() > result.getSpecial()) {
135.                result.setSpecial(ruleConf.getSpecial());
136.            }
137.
138.            if (!ruleConf.getSpecialChars().isEmpty()) {
139.                result.getSpecialChars().addAll(ruleConf.getSpecialChars().stream().
140.                        filter(c -> !result.getSpecialChars().contains(c)).collect(Collectors.toList()));
141.            }
142.
143.            if (!ruleConf.getIllegalChars().isEmpty()) {
144.                result.getIllegalChars().addAll(ruleConf.getIllegalChars().stream().
145.                        filter(c -> !result.getIllegalChars().contains(c)).collect(Collectors.toList()));
146.            }
147.
148.            if (ruleConf.getRepeatSame() > result.getRepeatSame()) {
149.                result.setRepeatSame(ruleConf.getRepeatSame());
150.            }
151.
152.            if (!result.isUsernameAllowed()) {
153.                result.setUsernameAllowed(ruleConf.isUsernameAllowed());
154.            }
155.
156.            if (!ruleConf.getWordsNotPermitted().isEmpty()) {
157.                result.getWordsNotPermitted().addAll(ruleConf.getWordsNotPermitted().stream().
158.                        filter(w -> !result.getWordsNotPermitted().contains(w)).collect(Collectors.toList()));
159.            }
160.        });
```

Figura 14 - DefaultPasswordGenerator.java {merge()} analisi con JaCoCo



```
75.    protected List<PasswordRule> getPasswordRules(final PasswordPolicy policy) {
76.        List<PasswordRule> result = new ArrayList<>();
77.
78.        for (Implementation impl : policy.getRules()) {
79.            try {
80.                ImplementationManager.buildPasswordRule(
81.                        impl,
82.                        () -> perContextPasswordRules.get(impl.getKey()),
83.                        instance -> perContextPasswordRules.put(impl.getKey(), instance)).
84.                        ifPresent(result::add);
85.            } catch (Exception e) {
86.                LOG.warn("While building {}", impl, e);
87.            }
88.        }
```

Figura 15 - DefaultPasswordGenerator.java {getPasswordRules()} analisi con JaCoCo

Figura 16 - DefaultPasswordGenerator.java analisi con JaCoCo

```
36.    public static Optional<Pair<String, String>> parseGroupOwnerRealm(final String input) {
37.        String[] split = input.split("@");
38.        return split == null || split.length < 2
39.                ? Optional.empty()
40.                : Optional.of(Pair.of(split[0], split[1]));
41.    }
```

Figura 17 - DefaultPasswordGenerator.java {parseGroupOwnerRealm()} analisi con JaCoCo

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	97%	68/70	36%	23/65	36%	23/65

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.syncope.core.spring.security	1	97%	68/70	36%	23/65	36%	23/65

Report generated by PIT 1.14.1

Figura 18 – Report Pit DefaultPasswordGenerator.java

Pit Test Coverage Report

Package Summary

org.apache.syncope.core.provisioning.api.utils

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	68%	27/40	78%	14/18	100%	14/14

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
RealmUtils.java	68%	27/40	78%	14/18	100%	14/14

Report generated by PIT 1.14.1

Figura 19 – Report Pit RealmUtils.java

Figura 20 - AuthDataAccessor.java {buildAuthorities()} analisi con JaCoCo