the BLACK BOX DATABASE

James Gamber

microneering, Inc.

Table of Contents

Introduction....1
Database Challenges....1
Overview....2
  Feature Absorption....3
  Distributed Processing....4
    The Eight Fallacies of Distributed Computing....4
Database Application Life Cycle....5
Database Design....6
  Knowledge Management aka Documentation....7
  Functional Design....7
  Detail Schema Design....8
Security....9
  Security Users....9
  Application Implemented Security....9
  SQL Injection....10
    Defensive Design Patterns....10
Configuration Management....12
  Release for Production....12
BBD API....13
  API Example....14
  Revised API....16
  Deprecated API....17
  API Repository....17
  API Design....18
Testing BBD API....19
Development, Test and Production....20
  Locking Down the Database....20
  Transition Development to Test....21
  Transition Test to Production....22
Object Persistence Frameworks....23
  Object Persistence....23
  Closure of Instances....24
Java Persistence Architecture(JPA)....25
  Ordinal Entity Class....25
  Entity Manager....25
  Life Cycle....26
Reference Examples....27
  Java MySQL Example....27
  Example Application....27

Example Classes....27
References....28
Appendix A – BBD Functional Requirements....29
SQL....29
Database Features....29
Deprecation....29
Validation....29
Security....29
Reference Implementation....30
Testing....30
Appendix B – Detailed Design Java/JDBC Implementation....31
Schema....31
Database....31
Tables....31
BBD Interfaces and Classes....32
BBD Java Interfaces....32
BBD Java classes....33
API....33
Lists....33
Connecting....33
Default Implementations....34
POJO Interfaces and Classes....34
POJO Class....34
BBD POJO Class....34
Connection Design....35
Limiting JDBC....35
Broker Design Pattern....36
BBD SWING....36
BBD Properties....37
Testing....38
API Unit Test Table....38
Test Table....38
Test Table API....38
Appendix C – Example Implementation BBD API....40
GetAPI....40

Introduction

Modern applications integrate with databases using an N-tier architecture. The overall system is comprised of many layers. The database can be thought of as being at the bottom of this architecture, and perhaps a web browser presenting information is at the top. In between can be found HTML, XML, JSP, ASP, web servers, networks, class libraries, virtual machines, message queues, etc.

Black box database(BBD) design constrains schema references to the database. The BBD exposes a consistent and rigid Application Programming Interface (API). The API of the database becomes as fixed as that of class libraries. In the case of class libraries, method calls may be deprecated, and marked as such, being replaced by newer and better methods and interfaces. BBD makes use of deprecated and revised API elements.

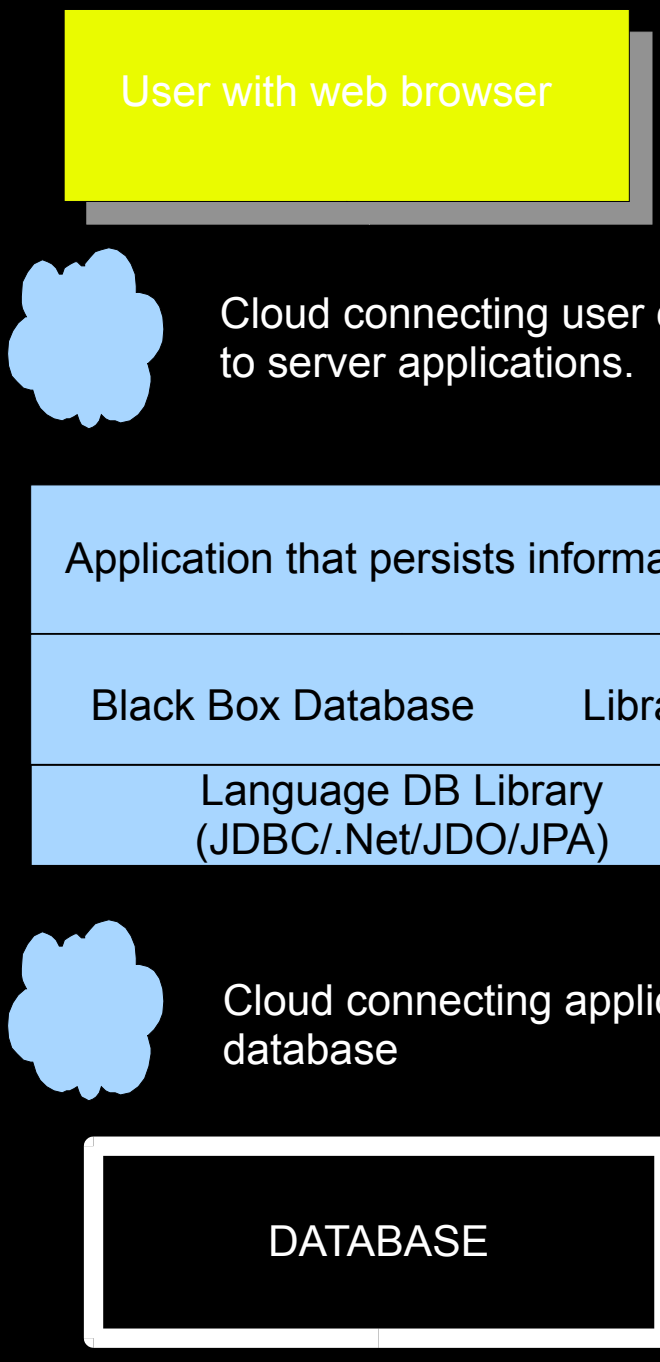
BBD ties requirements to schema and testability; testability to performance and security. No longer are these features buried and possible bypassed, but become and integral part of the design.

Database Challenges

Almost all databases end up with similar challenges after several years of operation.

- There is no traceability of history leading up to current configuration of schema.
- There are no requirements documents relating to the current schema.
- There is no testability against requirements.
- Performance deteriorates.
- Security deteriorates.
- Application layers that rely on the database fail sporadically.
- Minor changes in database schema have unpredictable and even disastrous consequences.
- New software developers have extended learning curves.
- Porting data between versions is moving from one unknown to another unknown*.
- *Its origin and purpose still a total mystery.

Overview



Feature Absorption

The application frameworks and layers outside the database frequently implement some of the same features already found in the database.

Database schema elements have been placed in all possible application layers. Frequently table and field schema identifiers, as well as passwords have been exposed in HTML. This is a great source of information to hackers. When the underlying schema changes, these layers are prone to failure. Often the failures are missed in testing and are found on the net by users.

Dynamic SQL is a number of software engineering techniques, that create SQL as a string then pass that string to the database for execution. These methods reference portions of database schema in one or more layers of the n-tier system. They produce a sequence of SQL commands pro grammatically at run time.

These means that the SQL is not inspect able or testable, since it does not exist until it is created. This makes it infinitely variable at run time, so its performance and security characteristics change continuously. Tuning is impractical, and performance quickly deteriorates in a database bombarded by these commands. Each new SQL string requires a new execution plan and a slow compile step to convert the SQL string to executable database instructions*.

Features that already exist in the database must be re-invented to support schema outside the database. Once there is schema, there arises the need for schema manipulation language like variants of the SQL, such as JDOQL.

As a general principle:

If the database implements a feature, don't re implement it again in another layer:

1. Don't re-implement SQL
2. Don't re-implement transactions
3. Don't re-reference schema elements by name
4. Don't re-implement replication

* I think you know what the problem is just as well as I do.

Distributed Processing

At the core, distributed processing scenarios raise a host of conflicting requirements. Is it better to limit the database information to only one realm, like a J2EE application server? Or should data be shared in a consistent enterprise manner across reporting, web, and legacy applications?

Keep database functions in the database: don't spread them around the landscape; don't reinvent them. Industrial strength databases have a variety of functionality that they implement extremely well.

Databases and their relationship with their external tentacles may become so inscrutable, that one DBA when asked about normalizing a database replied: "It is better to curse the darkness than to light a candle." When an installation reaches this point, it is time for BBD.

The Eight Fallacies of Distributed Computing

(by Peter Deutsch)
Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences*.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

* It can only be attributable to human error.

Database Application Life Cycle

The BBD life cycle includes the following:

- Documentation
  - Requirements
  - Design
    - Functional
    - Detailed
- Development Database implementation
- Development Database constraints
- Development Database unit tests
- Configuration Management of Schema, test programs, application programs
- Deployment and test in test database
- Performance review
- Security review
- Deployment to Production
- Automated production tests
- Repeat for next upgrade/version/release

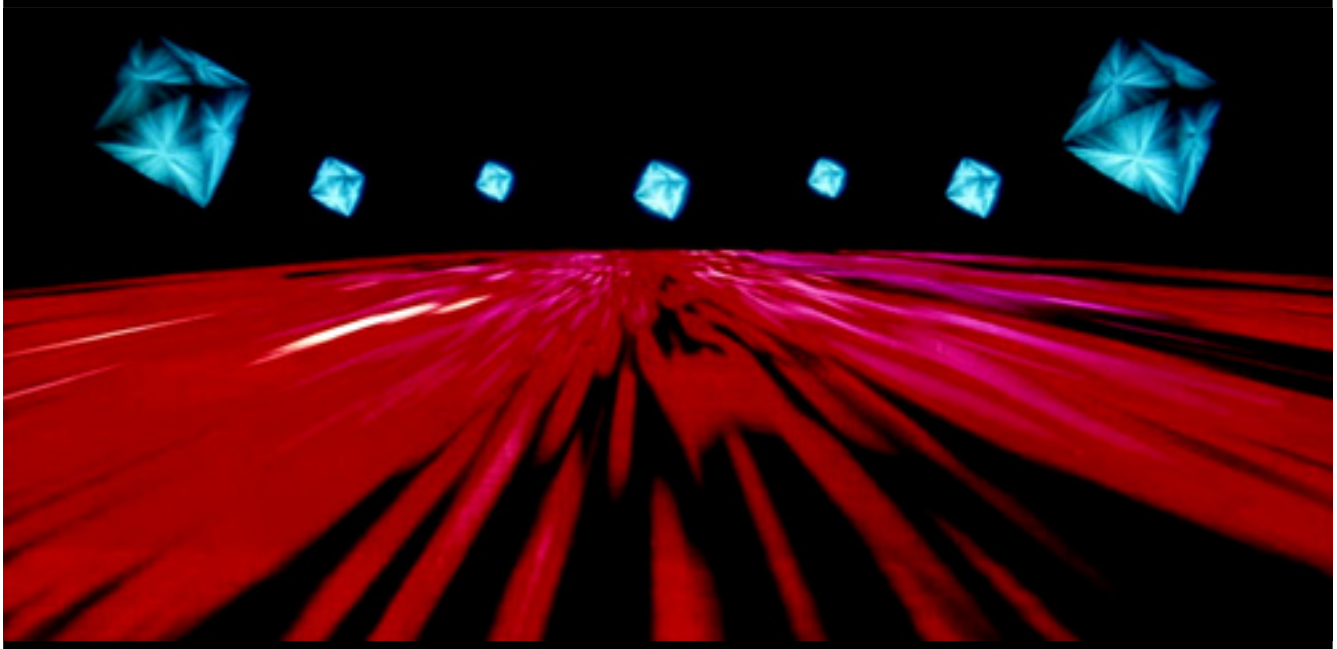
Database Design

It is very rare to find databases that are designed. A better explanation of the state of most corporate databases would be the term database accretion.

The normal process is to add new features as they are needed. Typically, the new feature is communicated verbally or in some transient document (the famous napkin or envelop). The database changes are often done by the programmer or DBA issuing Create or Alter statements directly into the database. The information they had, linking the new function to the schema changes, is soon lost.

Why is little or no attention paid to designing databases, when fortunes are spent and libraries of books are written on software design. The same rational that describes the life cycle costs of software are directly applicable to databases.

Over a period of months and years, the changes accrete, until they expand to cover the horizon.



Knowledge Management aka Documentation

BBD documentation are living documents. As requirements change, the documentation changes. As new technologies are brought into usage, the documentation changes.

Documentation is targeted at specific audiences. Requirements documents explain the business functionality, are excellent for: users to create expectations based on the intent of the system; developers to created designs that implement the features; testers to validate that the required features are delivered.

Effective documentation for work groups is best maintained on a web site that is structured and organized for fast search and retrieval with web browsers. The documents may be created in word processors and then stored as both word processor documents and exported as HTML documents.

In the best environments, documentation is generated at various levels for various audiences by the stake holders. The stake holders generate a series of Template documents with prescribed outlines, purposes, and audiences.

Since stake holder often lack technical writing skills, documents may be edited and distributed to a web server by the team's professional technical writer.

Documents are circulated among the stake holders of the business areas that will use the database.

Documentation answers the following:

- Why is this specific data persisted?
- Which business functions will make use of this information?
- What special algorithms are needed in the business process?
- How does this information combine in differing business processes?

Functional Design

The high level overview documentation is excellent for end users to specify the results that are expected at the other end of the pipe, sometimes a web site accessed by a browser. For example, a new customer contracts system might replace old manual procedures and paper documents. Many components of manual system might be persisted in the new systems database. Special security and performance expectations may be specified.

Detail Schema Design

Detailed schema design documents describe the tables and fields of the database, connecting these entities to the functions they implement. Indexes, keys, constraints, other similar internal schema information are described here in detail.

A variety of database tools create graphic presentations of table schema that is suitable for incorporation in detailed design documents.

Security

BBD implements a Root/User model of security. The BBD layer has its own schema/tables and login credential. This corresponds to the root mode. BBD uses its database objects to control the interface between application programs and the database. The root connection string, user/password is specified in the BBD properties file. This is the database user/password that accesses the BBD schema/tables.

An option exists to use root for all access. This might be a good mode during database and application development, but is discouraged for QA test and final production. In production, specific user credentials are prescribed for each application BBD API.

Unlimited specific user credentials may be created for access to one or more BBD API methods. Each BBD API method corresponds to one stored procedure. Applications may prompt the user for credentials, then pro-grammatically assign them to specific API methods.

The BBD credential is encapsulated in the BBD Principal class and is associated with a specific BBD API and connects (or login) to the database. The database ultimately controls user access with standard database conventions for assigning execute grants. An execute grant says: “This user may execute this stored procedure.” The DBA would not assign any application user with grants on objects in the BBD schema.

Security Users

Three users are provided in the sample implementation. User 'bbd' is root access. The 'bbd' user only has execute grant on the 'BBD' database.

Then user 'test' is used to run the unit tests. Since both 'BBD' and 'TEST' databases are tested with unit tests, user 'test' has execute grants on both database. SQL scripts for creating the users and grants are provided.

User 'Duke' has user execute grant to the example production database called 'TEST',

The development of security for BBD shows the effectiveness of unit tests. Initial software changes, to implement the BBD Root/User security model, caused the unit tests to fail at very specific places. These bugs were quickly identified and the sample implementation was quickly debugged.

Application Implemented Security

Once again, why create endless application software to do what the database does. The database has perfectly good authentication and authorization for users, including integration with Active directory. Don't worry about having 10,000 daily users, AD routinely works for 100,000 daily users. Separate Active Directories can be used for corporate and web users of your databases.

SQL Injection

An security threat to any database is SQL Injection. BBD design is has the advantage of minimizing this threat. The first principle in reducing the threat is remove all SQL from all layers above the database. Most web applications are riddle with the following type of code:

```
String sql = “SELECT * FROM users WHERE name = ' " + userName + " '; "
Resultset rs = connection.exec(sql);
while (rs.next) {
   out.print(“<b>rs.getString(\“Name\”)”);
}
```

This is the classic example of SQL Injection where the attacker simple enters a userName of

```
a' or 't'='t
```

This attack results in all user names in the database printed to the attacker's browser screen. With little more effort, the attacker can learn a huge amount about the database schema and create many SQL that pull information that the database owner believes to be secure and hidden.

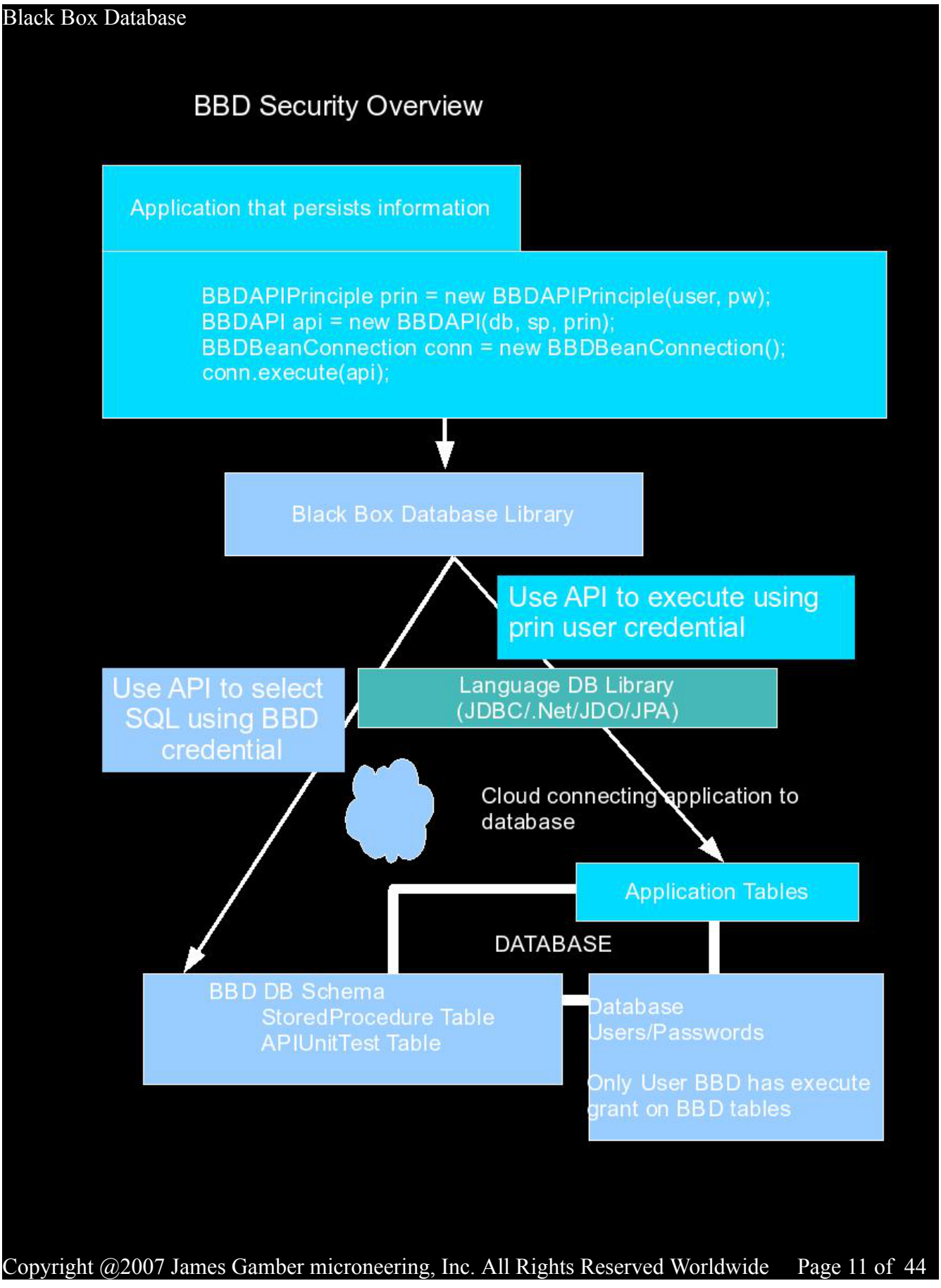
Defensive Design Patterns

The first defense against the security flaw, eliminate all SQL from the layers above the database. Designs based on moving SQL to the application level to make it database independent, usually end up making the attacker's methods database independent. BBD is the most comprehensive design pattern possible creating applications resistant to SQL Injection.

The second defense is use bind variables. These over come the immediate problems found in string concatenation shown above. BBD uses bind variables by default for all SQL.

The third defense is validation all user input. BBD has validation built in to the sample implementations. This validation uses compiled regular expression that automatically check all data types before binding the data to prepared statements and before execution of the SQL. The simple validation provided is extensive by data type and is easily extended to prevent escaped sequences and other SQL Injection techniques.

Unit tests the cover SQL Injection methods are easily devised and become a permanent on going check of your database security.



Configuration Management

Database elements may be manually created during design and development using one of many tools available. However, all databases have the capability to export the SQL needed to recreate schema elements: tables, fields in tables, indexes, keys, etc.

Once the development database has reached a stable release point, such as an alpha or beta build, then the database tools should be used to export the schema as specific text files containing SQL CREATE, ALTER, and DROP statements, referred to as Database Definition Language(DDL).

These text files are checked in to a configuration management (CM) system, such as Subversion. Using this technique, the complete history of schema changes can be connected to business requirements changes.

Never continue the ad hoc process of manual DDL changes into production. Never rely on database dumps and backups as your repository for DDL.

Release for Production

A principle of BBD: production database schema is not created or modified using the various ad hoc tools. Production database schema is always created and modified using the DDL SQL checked out from a configuration management system.

The production schema modification DDL have the following characteristics:

- DDL originates from CM system.
- DDL successfully modifies the test database.
- Documentation updates reflect the changes: requirement, functional, and detailed.
- New API tests are created.
- All API tests have been executed on the test database.
- DDL reviewed by the security authority conforms to site security requirements.
- DDL reviewed by the performance authority conforms to site performance requirements.

This is in stark contrast to the standard operating procedure in almost all database installations:

- Changes are applied manually with no particular procedure or traceability.
- If things go too terribly wrong, the database must be restored in total from its backup file. Some information may be lost or may be recovered with much effort.
- The impact of changes on database security and performance is almost never considered or tested before production changes are introduced.
- Results are predictably unpredictable.

BBD API

The API of the BBD is in the form of stored procedures. This is similar to the method signature of object oriented classes. Some databases, such as Oracle, additionally provide for method overloading for stored procedures with the same name, but different argument types.

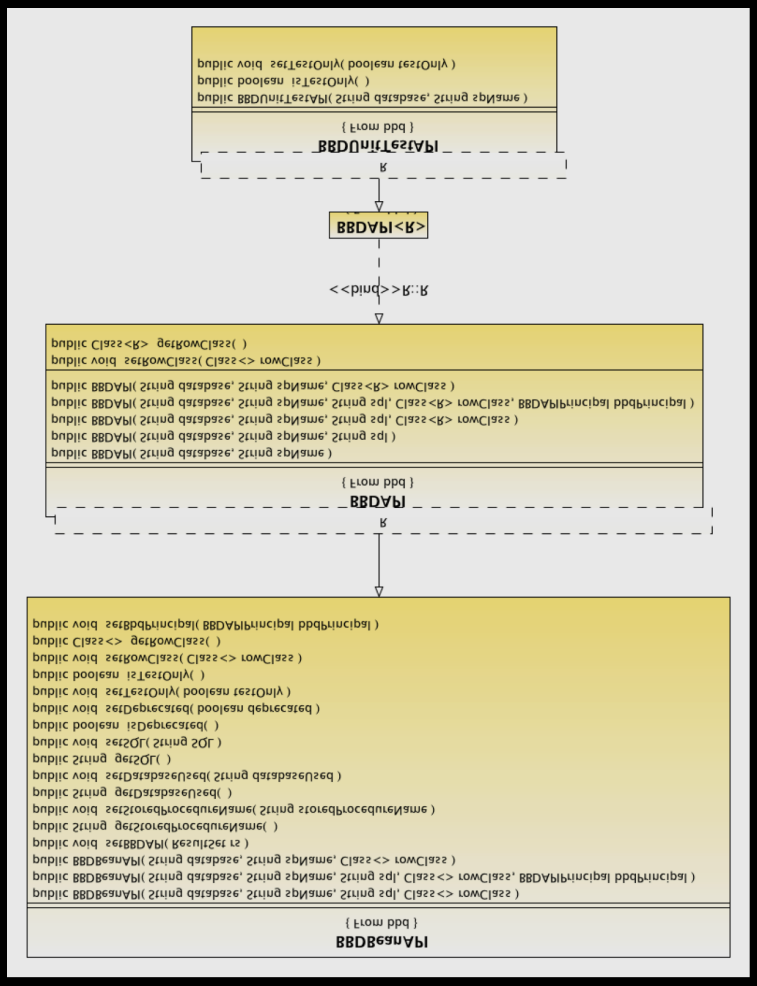
The BBD API stored procedures have the following characteristics:

- Stored procedures input output parameters, return values, and exceptions are invariant
- Stored procedures may be deprecated
- Store procedures may be replaced by revised versions with a different set of parameters, return values, and exceptions under a revised procedure name: getInventory replaced by getInventoryV2; or getInventory replaced by getInventoryAlertMinimum.
- The API may be extended by adding new stored procedures.
- Changes to schema result in changes to the internals of stored procedures. For example, if a returned value used to come from tableA and now comes from TableB because the schema changed, the stored procedures will still return the same value, but will get that value from the new tables as needed. The schema change is not exposed to the user application. The API is invariant to schema changes; except for deprecation, deletion, or addition of stored procedures.
- Specific user/groups are given execution grants of stored procedures. This indirectly provides them with ability to access or modify the database data.
- No access to tables is granted.
- Stored procedures are executed under specific user credentials, not a single generic user.

API Example

The BBD API provides all database access.

In the example implementation, a base clase BBDBeanAPI is provided.



The Bean API is used with the BBD JPA example implementation, and supports Java beans of any class. The derived class BBDAPI, is designed to support wrapper classes bound by the semantics of Java generics, such as TestDataModelRow.class show below.

The BBDAPI class specifies the API to the application program:

```
BBDAPI SELECT = new BBDAPI("test", "TestSelect",
                           TestDataModelRow.class);
```

This specifies the database(“test”), the stored procedure "TestSelect”, and the Java class containing the data model for each row of the result set.

The application programmer uses the following idiom:

- Create a stored procedure for each type of transaction on this table: select, update, insert, etc.
- Make an entry in the “storedprocedures” table providing a unique name for each stored procedure. The name can be any string, preferable more applicable to the business function than the schema details.
- Create a BBDAPI object for each stored procedure. This object is used in the application program as needed to interact with the database.

As we will see later, this idiom also involves at least two sets of unit tests, one to test the database by itself and a corresponding set of tests driven via the application program interface, BBDAPI. After these test are in place you merrily go about you chores of adding as many tiers to your new application as you like. Code away at your STRUTS, JSF, HTML, and AJAX. When your code isn't working, just run the unit tests.

It is amazing the types of things that come out of unit tests. One morning my application wasn't working, so a quick call to the DBA's who malevolently assured me I was quite mad and that Oracle was running perfectly*. However, after providing them with the ORA-xxxxx message from the unit tests, they determined that the rollback areas had filled up.

Unit Testing the database, independent of any business applications, saves untold hours of analysis and problem identification. Martin Fowler has summarized unit testing as follows: "Never in the field of software development was so much owed by so many to so few lines of code".

The BBD API documentation, Appendix C, describes the type of values to substitute, valid values / ranges, row sets returned, exceptions expected in the reference implementation.

Since unit testing is a component of BBD design, the core package contains the BBDUnitTestAPI. This API automatically records unit tests results in the database.

Revised API

A revision to an API is a change to the method signature, returned values, exceptions, or impact on stored data. To implement the revised API, the additional steps are taken:

The original stored procedure is marked as deprecated in the StoredProcedures table.

The revised stored procedures is named to indicate version. This could be a simple suffix to the stored procedure name, for example Original_Name_V2.

A new row is added to the StoredProcedures table for the new API.

* I can see you're really upset about this. I honestly think you ought to sit down calmly, take a stress pill, and think things over.

Deprecated API

Stored procedures may be marked as deprecated. Using a deprecated API results in warnings to the log files during Unit Testing and running application tiers. The source code that uses a deprecated API is identified in the log file with a stack trace showing the call stack leading up to the use of the deprecated API.

Once the source code is corrected to use the new API, the original Stored Procedures may be removed from the database. The DDL for obsoleted APIs will still reside in the CM system for future reference.

API Repository

In the reference example, a database table named 'storedprocedure' contains all the 'call' statements. This table is accessed by a seed API.

An example of a seed API would be: getAPI(?,?). The two substituted values are:

1st Parameter: database containing the stored procedure

2nd Parameter: stored procedure name

By calling the seed API, all stored procedures are accessed. This bottleneck is a convenient location to implement additional control and security*. Also, BBD uses this bottleneck to: cache API's and their metadata; ascertain deprecated or test status of the API; validate call arguments; generate logs; etc. The advantages of a bottleneck for databases are similar to those advantages of MVC2 over the MVC design pattern for STRUTS. Introducing a bottleneck for database access is a new design pattern:

D = M2VC2

* I know you and Frank were planning to disconnect me, and I'm afraid that's something I cannot allow to happen.

API Design

If the database has an API, then that requires an API design. Designing great APIs is fun and challenging, no less for databases than programming languages. See the references section for a great presentation by Joshua Block on Good API Design, and Why It Matters. Here are some of Block's main points on API design.

- Gather Requirements
- One Page Specification
- Drafts of API
- Aim to Displease Everyone Equally
- Expect to Evolve the API
- The API should be as small as possible, and no smaller
- Implementation should not impact API
- Minimize accessibility to everything
- Names Matter – API is a little language
- Documentation matters
- Consider Performance Implications of API Decisions
- API Must Peacefully Coexist with Platform
- Do make the application do anything the database could do
- Don't violate the Principle of Least Astonishment
- Fast Fail, Report Errors as soon as possible after they occur
- Throw exceptions to indicate Exceptional Conditions

Testing BBD API

The BBD is a testable, repeatable black box.

- Prepare written test plans that link functional requirements, detailed design, and test procedures.
- At least one unit test developed for each stored procedure.
- Automatically record test metrics in a database.
- Standard reporting provides XML or HTML formated reporting that summarizes test suite statistics and individual test failures.
- Unit test run automatically each day and are designed to restore test information so that the database returns to its original state on completion of each test.
- Unit Tests are run against development and test databases.
- Performance Tests are run against production databases.
- Performance Tests* run during peak and sensitive response times of day.
- Trend checking of run times automatically flags stored procedures that do not meet required response times or that are slowing down over time.

Each API stored procedure can be individually tested using JUNIT. In the reference examples included Java/JUnit and .Net/nUnit unit tests are provided.

Unit tests may require test SQL when testing a database. This SQL is flagged as 'Test Only'. This flag may be used to remove test SQL from production.

Test metrics are saved for each test in APIUnitTest table of the database. From here it can be reported and analyzed using a number standard tools. Capture of metrics is facilitated by extending the abstract class APITest.

* I am putting myself to the fullest possible use, which is all I think that any conscious entity can ever hope to do.

Development, Test and Production

It is important to make a distinction between development, test, and production databases.

During the design and development phases, the software engineers should own and have full control of their own development database. They are free to write SQL like “select * from employee” or any other SQL they chose and they are free to place it in any layer of the overall system. It is necessary to free development databases from any constraints, so that the design can evolve into a final form as rapidly as possible.

However, for BBD design, the developer knows that when the design is completely implemented, it can not contain SQL like “select * from employee”! In the final stages of design all schema references are pushed back into the BBD and hide behind the API; database constraints are tightened to maximum extent using foreign keys and removal of all table access privileges.

Locking Down the Database

Locking down a database to prevent the emission of schema information is very simple. This is done by simply not granting any table level privileges. Simple SQL statements like “select * from employee” do not work without someone granting some user access to the employee table. Without this access grant, the SQL simply fails.

Transition Development to Test

There are several important steps when moving systems from development to test databases.

- All application source code for test comes from source control system.
- All schema, stored procedures, and DDL for test comes from source control systems.
- All data comes from controlled storage areas. DBUnit has an approach to storing test data in XML and using it to initialize tables with with known test data.
- Test database is locked down, see above.
- Constraint testing: Any missing constraints are identified, documented, applied on the Test database. Any changes are bug reported. DDL is generated and added to source code control.
- Performance testing: Any needed performance options, like indexes are identified, documented, applied on the Test database. Any changes are bug reported. DDL is generated and added to source code control.
- UseTestAPI property of BBD is set “true” to run unit tests.
- SQLPassThrough property is set “false” to identify any dynamic SQL in system. This SQL is replaced by stored procedures. Any changes are bug reported. Changes are added to source code control.
- Bug reports are only closed after source control is updated.

The test database is the best place for performance testing and security policy analysis.

Transition Test to Production

Moving systems to production is the occasion for heightened alertness and adequate contingency planning. If possible, apply highly transactional systems to subsets of users. For example, roll out to 10% of the user base on day 1; 50% on day 2; 100% on day 3. Have rollback procedures that are engineered to rapidly flip system back to their prior versions.

- Prepare production roll out plan document.
- All system source code for production comes from source control system.
- All schema, stored procedures, data for production comes from source control systems.
- Production database is locked down.
- UseTestAPI property is set FALSE.
- SQLPassThrough property is set FALSE.

Object Persistence Frameworks

This book implements two frameworks: for Java the Java Persistence Architecture(JPA); and for .Net the SourceForge ObjectBroker.

There is a clear demarcation between Object Oriented (OO) languages and traditional databases. Databases are designed to share data, manipulate rows sets, and maintain data integrity. OO design has grown up as a response to requirements of large coding projects, where more than 500,000 lines of code overwhelmed the capabilities of most software developers. OO was needed because these large projects simple collapsed under their own weight without it.

BBD object persistence framework is the traditional type of layer between the application and the database, intended to reduce the plethora of options available for the application/database interaction.

Several issues are explored in these implementation, while other features must be abridge because they conflict with BBD design principles. The implementations herein are purposefully not intended as complete implementations of either JPA or ObjectBroker. The frameworks implemented rely on BBD design to achieve testability, greater performance and robustness.

Object Persistence

BBD is about all types of database interactions, not only object persistence. Often application design encapsulate information from the database into OO classes representing entities useful to the specific application problem space.

Object oriented applications can have massive numbers of objects that vary greatly in size, complexity, and the way they reference one anther. Persisting these objects in a simple, effective, transactional manner is the goal. Examples of these are found in J2EE, JPA, JDO, etc. These techniques usually incorporate features that already exist in the database (feature absorption).

Database relationships that OO systems might attempt to maintain:

- One customer to many orders.
- Many employees to one department.
- Many orders to many products.

Closure of Instances

A Customer object may referenced departments and the departments may referenced the company. The company may have references that eventually include the entire database.

A group of objects that reference each other is called an object graph. Object graphs can be fairly large, as in the graph of objects reachable from Customer.

The persistence-capable class may have some fields that are themselves persistent, and other fields that are not. The closure of all persistent instances referenced from any one is the object graph arising from traversal of persistent field references.

Java Persistence Architecture(JPA)

JPA provides a standard for persistence of objects in databases through either annotation or XML configuration files.

Ordinal Entity Class

Some of the key annotations are:

@Entity – defines the Java class as persisted

@Id – defines the class variable as the database table key for finding a specific object of this class.

- The primary key class must define equals and hashCode methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.
- A composite primary key must either be represented and mapped as an embeddable class or must be represented and mapped to multiple fields or properties of the entity class .
- If the composite primary key class is mapped to multiple fields or properties of the entity class, the names of primary key fields or properties in the primary key class and those of the entity class must correspond and their types must be the same.
- The application must not change the value of the primary key. The behavior is undefined if this occurs.

With these two annotations, a simple class can be persisted. Such a class contains no non-transient class members that refer to persisted objects.

Entity Manager

The BBD Entity Manager is a subset of JPA that is capable of persisting an Ordinal Entity class type object.

Persistence.createEntityManagerFactory

```
@PersistenceContext EntityManager em;
Customer cust = em.find(Customer.class, custID);
```


Life Cycle

http://cayenne.apache.org/doc12/persistent-object-lifecycle.html

You can register callbacks for the following lifecycle events

- PrePersist
- PostPersist
- PreRemove
- PostRemove
- PreUpdate
- PostUpdate
- PostLoad

The only other rule is that any method marked to be a callback method has to take no arguments as input, and have void return.

Reference Examples

Black Box Database concepts may be applied to any database and application development environment.

Two reference examples are included, one for Java and one for .Net.

Java MySQL Example

Below is an example implementation using MySQL, Java, and Java Database Connectivity(JDBC).

Example Application

MyApp uses a Data Model Broker pattern where the application connects to the database via Broker class. The broker class encapsulates the database connection, interaction, and error recovery.

MyApp uses a Data Model Row pattern is used to hold the data returned be each row of a result set, or each row to be inserted/updated.

Example Classes

Data Model design pattern classes:

class HelloDataModelRow extends BBDDefaultRow

class HelloDataModelBroker<E extends HelloDataModelRow,F extends BBDRowArrayList<E>> implements BBDBroker<E,F>

POJO user created bean classes may be of any type.

public class HelloBean

public class HelloBeanBroker<E extends HelloBean>

References

http://www.codinghorror.com/blog/archives/000117.html

http://www.jpox.org/docs/1_2/jdovsjpa.html

http://www.jpox.org/

http://www.orientechnologies.com/docs/JavaDataObjects-RobinRoos-1.0.pdf

JSR 220: Enterprise JavaBeans™ 3.0

API Design

2001: A Space Odyssey

SQL Injection

Appendix A – BBD Functional Requirements

SQL

BBD will restrict database access to execution of stored procedures. The function signature of the stored procedure is referred to as the API.

No SQL will be used in the BBD layer.

The use of SQL from the Application layer will be an option. This option allows SQL use during development to prototype, then provides for switching it off for QA and production. BBD will report the location of SQL usage by stack trace to the originating source code line.

API access by the application layer will be by referencing the API database and API name through the BBD layer.

Database Features

The principles behind eliminating SQL are more general in nature. All features implemented by the database, for the database, will remain in the database. These features include: transactional processing; row set fetch sizes; integrity constraints; result set scrolling.

A great example of an unwieldy, fragile, and slow design is ADO row set updating. SQL traces show that updating one field deletes the entire row and reinserted it, some of the most expensive operations possible on a database.

Deprecation

BBD will provide for API deprecation. Optionally, deprecated API's may be turned off. Location of deprecated API's will be logged by stack trace to the originating source code line.

BBD will provide for API caching. Caching will be be reset periodically to allow for reloading of schema changes. This reset will occur during normal production usage with no interruptions to production.

Validation

BBD will automatically validate parameters passed to the API before the API is executed on the database. Parameter validation will be dynamic based on meta data retrieved from the database and the class type of the parameters. Parameter validation will use compiled Regular Expression.

Database requested will be validated against cached list of databases.

Security

A root/user model of security will limit access to BBD control database. Users will be created for

specific application API's as needed by application developers. Optionally, the application may use the root user for all application access, thereby eliminating the need for user credentials.

Authentication will be done by Operation System or database standards, not by BBD. Authorization will be done by Operating System or database standards, not BBD.

No grants of access will be provided for tables. All tables will be accessed by providing grants to specific API's.

BBD configuration properties will reside in a single file. The configuration will contain the credentials for root and unit test users.

Reference Implementation

Two sample Applications will be provided, one in Java/JDBC/MySQL and one in C#/.Net/SQLServer. Reference implementations will use a broker design pattern to encapsulate API BBD references.

SWING components will be provided with high level access to database CRUD funcitons.

Testing

Unit tests will be provided for the BBD layer

At least one additional table will track unit test. This table will persist date, duration, unit test method, number of rows selected, updated, inserted, and deleted by the unit test.

At least one unit test will be provided for each API.

Unit tests will be provided for the sample applications.

APIs can be marked for use in unit tests only. These APIs can be reported on the error log if there is an attempt to execute in production.

Appendix B – Detailed Design Java/JDBC Implementation

Schema

Database

BBD - the database for storing SQL API

Tables

API Table

The StoredProcedure table contains the API. It has one row for each API that can be used by BBD and the applications. As new database functions are needed, rows are added to this table. Each row represents one stored procedure that will be executed by BBD.

This table has the following fields:

- Database - where stored procedure is executed.
- SPName – name used to refer to stored procedure from BBD.
- BBDSQL – SQL used to execute the stored procedure
- Deprecated – Flag indicating the stored procedure is deprecated
- Test Only – Flag indicating the stored procedure is used for test and should not be included in production.

Database and SPName form the unique primary key for the table.

BBD Interfaces and Classes

A Java interface is provided for data fields and rows from result sets.

BBD Java Interfaces

interface BBDField

One field from one row of the result set

interface BBDColumn

One column from the result set

interface BBDRow<E extends BBDField>

One row from the result set

One row for update, insert, delete

interface BBDBroker<E extends BBDRow, F extends BBDRowArrayList<E>>

Encapsulated BBD and API functions:

Set connection principal

Select list of rows, Insert row(s), Update row(s), Delete row(s)

BBD Java classes

API

BBDAPIPrincipal

Logon username password

BBDBeanAPI

Base API class for POJOs

Contains database and API name (key)

Contains class used to create POJO lists from rowsets

Contains test and deprecated flags

Contains user supplied SQL

BBDAPI

extend BBDBeanAPI for BBDRows

Lists

BBDBeanArrayList<E> extends ArrayList<E> implements BBDColumn

List of POJO rows generated from a result set

BBDRowArrayList<R extends BBDRow> extends BBDBeanArrayList<R>

List of BBDRow'ss generated from a result set

Connecting

BBDBeanConnection<B, L extends ArrayList<B>>

Connection for processing POJOs

BBDConnection<R extends BBDRow, L extends BBDRowArrayList<R>>

extends BBDBeanConnection<R, L>

Connection for processing BBDRow's

Default Implementations

Example default implementations are provided for interfaces.

BBDDefaultField implements BBDField

BBDDefaultRow<E extends BBDField> extends ArrayList<E> implements BBDRow<E>

POJO Interfaces and Classes

BBD implementation with plain old Java objects (POJO) is also provided. This interface associates any Java class with a row of information from the stored procedure. The BBD stored procedures may return rows that correspond to rows in a table, or they may return rows that are fabricated through complex schema relationships. The internals of the stored procedures do not matter to the POJO.

POJO Class

Use any Java class. The class should conform to Java bean criteria of an empty constructor and setter/getter methods for class variables.

BBD POJO Class

class BBDBeanList<E extends Object> extends ArrayList<E>

class BBDBeanConnection<E extends Object, F extends List<E>>

Connection Design

JDBC provides a standard approach to connecting to the database. The JDBC connection utilizes a user name and password type authentication to provide access to database resources. The user name and password are encapsulated in a BBDAPIPrincipal class.

The reference implementation shares connections for all API that use the same BBDAPIPrincipal. This greatly improves performance, Both MySQL and SQL Server current drivers support multi-threading and a large number of connections. The root connection remains open and is used for all references to the BBD database. Other connections are finalized by the GC when their last usage is finalized.

Once connected to the database, JDBC provides for queries and update functions. SQL statements processed on the connection may have values substituted into the SQL prior to executing the stored procedures.

Two connection classes are provided:

- BBDBeanConnection
  - List executeQuery(final BBDAPI storedProcedure, final Object... params)
  - int executeUpdate(final BBDAPI storedProcedure, final Object... params)

- BBDConnection extends BBDBeanConnection
  - BBDRowArrayList executeQuery(final BBDAPI storedProcedure, final Object... params)

Limiting JDBC

This limited implementation of the JDBC API is intentional. Other JDBC API functions are considered database functions by design and are implemented in stored procedures if needed. An example of this is row set limits and forward and backward scrolling functions. Obviously meta data functions are total off limits in the application layer. This example implementation is secure and sustains high performance.

Broker Design Pattern

The method calls for the database API are encapsulated in broker classes. A broker pulls together related API functions, for example all database interactions with the Inventory information would be placed in an InventoryBroker class. A Java interface is provided for brokers.

The broker example classes are not in the core BBD, but are found in the example programs.

HelloBeanBroker<B extends HelloBean>

Example processing POJO

HelloDataModelBroker<R extends HelloDataModelRow, L extends BBDRowArrayList<R>>

implements BBDBroker<R, L>

Example processing POJO as BBDRow

HelloDataModelBroker<R extends HelloDataModelRow, L extends BBDRowArrayList<R>>

implements BBDBroker<R, L>

Example processing TestDataModelRow extends BBDDefaultRow<BBDDefaultField>

BBD SWING

Several SWING examples are provided that demonstrate BBD as an excellent basis for persistence of data models for many swing classes.

BBDJTable<R extends BBDRow, L extends BBDRowArrayList<R>> extends JTable

Example JTable backed by BBD classes.

BBDCrudPanel extends javax.swing.JPanel implements

TableModelListener, ListSelectionListener

Example JPanel that displays rowsets and provides Create, Update, Delete functionality using BBD classes.

BBD Properties

A simple BDB.properties file is used to set run time options.

Specify the Java JDBC Driver

DriverClass=com.mysql.jdbc.Driver

Specify the JDBC logon connection string

ConnectionString=jdbc:mysql://localhost/BBD?user=root&password=

Specify the database containing the BBD API information.

BBDDatabase=BBD

Specify the seed SQL used to get all other API SQL

getAPI=call getAPI(?,?);

Specify if Dynamic SQL may be used

SQLPassThrough=TRUE

Specify if Deprecated API should be cached. When cached, only the first use of the API is logged as deprecated. When not cached, every use of the API is logged as deprecated.

CacheDeprecatedAPI=FALSE

Set this to false for production, this will prevent test API's from running.

UseTestAPI=TRUE

Unit Test logon credential

BBDUnitTestUser=Test

BBDUnitTestPW=Test

APIUnitTestInsert=APIUTInsert

Testing

API Unit Test Table

This table will contain the results of the units tests performed against the stored procedures (aka the BBD API) and contains the following:

- Time of execution
- Duration of execution
- Number of reads
- Number of inserts
- Number of updates
- Number of deletes
- Test Name

This table is populated automatically by every unit test run. This is accomplished by writing unit tests that extend the abstract class BBDUnitTestCase

BBDUnitTestCase<R extends BBDRow<BBDField>, L extends BBDRowArrayList<R>>

This abstract class must be used to write two unit tests for each API:

- One tests direct calls to the BBD layer for each API.
- One tests application context by calling all Broker methods for the API (or similar application defined data access objects(DAO design pattern)

Test Table

This Test table is located in a special Test database.
This table is provided for testing CRUD functions.
The test user credential is “Duke” user, “Java” password.

Test Table API

```
private final BBDAPI<R> SELECT = new BBDAPI<R>("test", "TestSelect");
private final BBDAPI<R> INSERT = new BBDAPI<R>("test", "TestInsert");
private final BBDAPI <R>UPDATE = new BBDAPI<R>("test", "TestUpdate");
private final BBDAPI<R> DELETE = new BBDAPI<R>("test", "TestDelete");
```



```
    static final BBDAPIPrincipal userAccess = new BBDAPIPrincipal("Duke", "Java");
```


Appendix C – Example Implementation BBD API

StoredProcedure Table

Database	SP Name	SQL	Deprecated	TestOnly
bbd	getAPI	call getAPI(?,?)	0	0
bbd	HelloWorld	call HelloWorld()	1	0
bbd	HelloWorld2	call HelloWorld2(?)	0	0
bbd	getDB	call getDB()	0	0
bbd	APIUTInsert	call APIUTInsert(?,?,?,?,?,?,?)	0	1
bbd	getTableNames	call getTableNames(?)	0	0
persist	createTable	call createTable(?,?)	0	0
persist	persistEntity	call persistEntity()	0	0
test	TestUpdate	call TestUpdate(?,?,?,?,?)	0	0
test	TestInsert	call TestInsert(?,?,?,?,?)	0	0
test	TestDelete	call TestDelete(?)	0	0
test	TestSelect	call TestSelect(?)	0	0

GetAPI

Description

Read one row from the StoredProcedure table.

Parameters

Database Name – input name of database containing stored procedure.

Stored Procedure name – input name of stored procedure in the stored procedure table.

Unit Test

Database – bbd.BBDAPITest.testUsingDatabase()

Application - myApp.myDataModel.TestDataModelBrokerTest()

