# METHODS & TOOLS

**Practical knowledge for the software developer, tester and project manager** **ISSN 1661-402X**

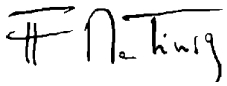
Fall 2013 (Volume 21 - number 3) www.methodsandtools.com

## Looking for Renaissance Software Development

As Agile is becoming a mainstream software development approach, I have seen on the Web more content about the importance of having cross-functional people in Scrum teams. Another denomination for this type of developer is to call them "T-shaped" people. The "T" symbolizes the deepness of the expertise in a particular topic and the broadness of knowledge in other areas of software engineering.

Large software development organizations have often chosen to have a tayloristic approach where people are separated according to their expertise: programmers, testers, DBA, business analysts. This separation in silos had also an implied "hierarchy" in the mind of developers: you are a tester or a business analyst only because you cannot code ;O) There is nothing wrong with being an expert and you cannot be expert in everything. This division, that was often increased by the fact that people were working in different offices, has created a "us versus them" mentality. I had the chances to work mainly in software development projects where I was able to perform different roles and I have found this situation beneficial. This is also because I have witnessed the loss of information when communication involved multiple people between the end-user and the software developer.

People like Leonardo da Vinci that were active in multiple fields from the art to engineering inspire the Renaissance reference in the title. We are not all geniuses like Leonardo da Vinci, even if some of us might think so, but we should be inspired by the broadness of his activities. Successful software applications are the combination of many elements that span from requirements to database performance and developers should know how they work together. This is why you find in our quarterly PDF issues articles that cover all the areas of software engineering. If you are a software renaissance developer, Methods & Tools is for you.



## Inside

# Experiential Learning: What Does it Have to Do With Agile?

Karen Greaves and Sam Laing, www.growingagile.co.za

Do you remember learning to ride a bike or drive a car? You didn't learn how to do it by reading a book, or attending a lecture did you? I am also sure you didn't get it right the first time you tried. You learned by trying it out, again and again, and slowly improving until you got to the point where you didn't even need to think about it anymore, it just happened.

This model of learning is called experiential learning. Essentially it means we experience something as a way of learning about it. It's very effective, yet in school we are programmed to learn by listening to the teacher. Unfortunately this lecture based learning style is still very prevalent today.

We use experiential learning as a key component of agile training. We find it far more effective than traditional lecture based learning. Agile is about learning how to do something rather than just passing a test. As a result, having an opportunity to fail and try again in the safety of a simulation using LEGO instead of business critical software can help teams grasp agile concepts much more quickly. It also makes them much more likely to remember the learnings once they encounter similar issues in the work place. In this article we will look at what experiential learning is, why it works, and how it is helpful for agile training.

**So what is experiential learning and why does it work?**

As it suggests in the name, "experiential learning is the process of making meaning from direct experience"; or "to discover and experiment with knowledge firsthand, instead of hearing or reading about others' experiences". [1]

According to David Kolb [2], an educational theorist, in order to gain genuine knowledge from an experience, certain abilities are required:

- The learner must be willing to be actively involved in the experience;
- The learner must be able to reflect on the experience;
- The learner must possess and use analytical skills to conceptualize the experience;
- The learner must possess decision-making and problem solving skills in order to use the new ideas gained from the experience.

We find experiential learning maps well to "Training from the BACK of the room". A training method developed by Sharon Bowman [3] based on how the brain works.

Sharon Bowman emphasizes some key points for helping people to learn.

- People learn in different ways and therefore it is useful to include a range of activities like moving, reading, writing, listening, drawing and talking into any training.
- Our reticular activating system keeps us sane by tuning out lots of stimulus. To make sure we aren't tuning out training, the training needs to change every 10 to 15 minutes.
- We need repetition to remember things. Sharon's advice is 6 times in 6 ways to remember something.
- People don't learn when they are afraid, so it is important to create a safe environment in training to allow people the best opportunity to learn.

**Okay if it's that great, why don't more people use it?**

We often look to experts to learn. In the agile community it is commonly these experts who are teaching others. However, being an expert in a field does not make you an expert trainer. Simply lecturing what you know is not training. Effective training is about understanding and application of that knowledge afterwards. As mentioned earlier in the article, most of us were exposed to lecture based training in school and university. It is commonly what we expect when we think about training. Many people have never experienced something other than training by PowerPoint.

**How have we used this?**

We use "Training from the BACK of the room" as a training style to incorporate experiential learning. There are some simulations or games we use regularly in training agile teams. Without fail we find that the insights people share when debriefing these games are far more powerful that any lecturing points we could teach. People often refer back to the game, when trying to explain why they should do something differently at work, a sign that the experience of the learning worked.

**Let's look at some examples**

- **Scrum Lego Simulation**

When training teams in Scrum we use a LEGO simulation [4] where teams have to build a LEGO Zoo in two 10-minute sprints. The most common lessons teams learn through the simulation are:

- The value of working on one thing at a time
- How tasks don't need to be assigned to individuals up front, it can emerge during the work
- The importance of having a shared understanding during planning
- The value of communicating regularly with your Product Owner and stakeholders

Teaching these points in a classroom, don't work half as well. Although people might understand the theory they do not have physical experience of how working in this way is better. In a relatively simple simulation, people try one way, fail, try an alternate way and succeed. This is a much better method for reinforcing behaviour than telling people why they should change.

- **The Airplane Game**

A favourite game of mine is the Airplane Game [5]. This is a powerful game because it physically shows something that is counter-intuitive. In this game, which looks at limiting work in progress as a way to speed up cycle time, people realise for themselves that being busy is not the same as being effective.

One of the reasons this game is so effective is that people build physical paper airplanes. Often in the world of software, since all our artefacts are electronic it is difficult to see work piling up at a bottleneck. Having a pile of paper planes accumulating makes the pressure surrounding it visible. People can literally see the bottleneck. Having made this visible, it is easy when debriefing for people to relate this to work, and realise how they are creating a bottleneck in a specific area. This same logic is why physical task boards work so well.

### Agile Testing with Jenga

A topic we have been doing a lot of training on recently is agile testing. We try to promote testing first through TDD and having testers and developers pair. Often teams like the idea but believe it will slow them down. We use a great game with Jenga [6] to help people realise that although testing upfront and as you go seems to slow you down, if you count the time it takes to correct the bugs that are found, this is much faster than traditional approaches.

### So what can I do?

Think about the last time you attending training? What was it like? And what do you remember from that training?

Next time you attend training, find out before hand a bit more about the trainer's experience and style. Also think about why you are attending and what you hope to learn. If the trainer does run an exercise, get stuck right in. The more you participate the more you will learn. If the training doesn't give you an opportunity, think about how you can practice what you have just learned the very next day, to get some experience and reflect on how it can help you.

Now think about the last time you tried to teach someone something new: Maybe to a colleague at work? Maybe show your grandparents how to use a computer? What was that like? How much of what you taught them did they retain? How much were you talking, and how much were they experiencing for themselves?

Next time you teach something, consider how you can get the person to experience the concept in a safe environment where they can fail and self-correct.

If you'd like to know more about creating experiential training, here are some resources we would recommend:

- Growing Agile: A Coach's Guide to Training Scrum [7]
- Training from the BACK of the room [8]
- Tasty Cupcakes [9]
- Collaboration Games [10]

### References

[1] http://en.wikipedia.org/wiki/Experiential_learning

[2] http://en.wikipedia/wiki/David_A._Kolb

[3] http://www.bowperson.com

[4] https://leanpub/TrainingScrum

[5] http://www.shmula.com/paper-airplane-game-pull-systems-push-systems/8280/

[6] http://nandalankalapalli.wordpress.com/2011/09/15/game-test-small-test-often/

[7] https://leanpub/TrainingScrum

[8] http://www.amazon.com/Training-From-Back-Room-Aside/dp/0787996629

[9] http://tastycupcakes.org

[10] https://leanpub/CollaborationGamesToolbox

TinyPM Smart Agile Collaboration Tool - Click on ad to reach advertiser web site

# Use the Debugger, Stupid!

Jaroslav Tulach, http://www.apidesign.org/

My name is Jaroslav Tulach and I am founder and initial architect of NetBeans [1], which is not just a well known IDE for Java, PHP, JavaScript, C/C++, HTML and who knows what else, but also the first modular desktop application framework written in Java. Maintaining the NetBeans Platform, its architecture and APIs has always been my primary focus. During more than sixteen years of participating in the NetBeans.org project, I have seen, made and helped to recover from many design mistakes. Such experience obligated me to sit down and summarize it in Practical API Design [2] and 20 API Paradoxes [3] books. Moreover I maintain a wiki at apidesign.org with online resources about designing APIs.

However today I don't want to focus on APIs, rather than that I want to share few observations about debuggers. The debugger is a program that helps us find out what other programs are doing. Often people use a debugger to understand why their own code does not do, what they would like it to do. However the debugger is incredibly useful for other purposes as well as you will discover in this article.

## Building Modular systems

Chapter 1 of the Practical API Design book discusses the art of building modern software systems and concludes that these days we are building systems that we don't understand. As an example it mentions that in order to write a dynamically generated web page one needs to understand HTML and possibly a Servlet API, but you don't need to have knowledge of anything that lays below: Glassfish server, Java VM, Unix libraries, Linux kernel, the VirtualBox nor even the hardware. At the end you can deploy a massive solution and in fact understand roughly one percent of it!

Some people may find it frightening that we use systems without real understanding, but The Art of Building Modern Software chapter concludes that such level of cluelessness is OK and in fact completely necessary. Brain has limited capacity, and the learning time is finite as well. As such we just can't understand everything. It is enough to know just the surface:

- the Servlet API - but not its actual implementation in the Glassfish server
- the user or command line interface of VirtualBox - to install Linux into it and to launch it
- the apt CLI - to install Java and Glassfish

Knowing these **interfaces** is enough. The details behind them are unimportant (if things go well).

By the way: this is also the reason why API design is important (command line parameters, as well as layout of files, are as important types of API as classes and their methods). Well-designed application programming interfaces encourage cluelessness. The less knowledge people need to have to use a technology, the better the technology is. That is why it is still useful to read my Practical Design Book and learn how to design APIs properly! But let's now get back to topic of debugging.

### Cluelessness of an API User

I am using word "cluelessness" for the blessed state when one does not need to know the internals behind APIs. However in fact I mean **selective** cluelessness: when there is no need to know to make things work, the less we know the better.

However when things go wrong, then we should concentrate on what is wrong and get as deep knowledge of the affected area as possible. We should **select** that we want to know everything about a particular part of Glassfish for example and learn every detail about it, for example why it throws some NullPointerException.

These days it should not be hard to increase our knowledge about our libraries. Most of them are open source and thus getting access to their source code is easy. Putting a breakpoint on appropriate line, launching our own application under debugger is usually easy as well. Moreover it is also easy to play with the code: modify a line, compile (using Maven or *make*) and use the modified version. There are no barriers to increase our knowledge about Glassfish, Java, Linux, etc.

In spite of the above easiness, I don't see people doing it. I don't see them looking at sources of libraries they use. Rather than that they treat them as blackbox. In spite of all the options we have, some of us decided to stay clueless (in the negative meaning, without the important **selective** part).

### The Googling Robot

Rather than trying to investigate the problem, the first reaction is to query the Google search engine. I have to admit, I do it too and the results from Stackoverflow web site and really valuable. It is *selectively* clueless to check whether somebody else tried to solve similar problem, so far this attitude is good.

However the really problematic state occurs as soon as the query yields no results. I've seen many situations when desperate programmers frustrated by unsatisfying search gave up! Such googling only robots will however face inevitable destiny. These days Google can recognize voice, drive a car - everything done by a brute force searches for correlations. The day when Google will be able to create artificial intelligence that will program using searches is not far away. At that moment programmers will either choose to go out of their previous business or will have to become *selectively* clueless again - and learn how to use debugger.

By the way: you may object that one day an Artificial Intelligence that knows how to debug will be created as well. I am sure somebody will try, but first of all such day is further away. Second, let us remind the foundation of computer science - the Halting problem - which shows that there can't be a program which would understand behavior of all programs - thus at least some need for human programmers will always remain.

### Knowing Every Detail

Once a customer of the NetBeans Platform invited me for a consultation. It is not common to send the creator of a framework for a consultation, the usual strategy is to send the junior guy to find out if they can spell Java. Later you send somebody more experienced and only if every effort fails, you ask the guru. But even for me, as a creator of the NetBeans Platform, it is important to see real customer from time to time, so I took the opportunity to visit this important customer.

Certainly I can talk about architecture decisions and explain how to use NetBeans APIs on a higher level, but when it comes to details, I may often be in total cluelessness situation. There is a lot of NetBeans APIs check at http://bits.netbeans.org/7.3/javadoc/ and we design them as a team. As a result my most common answer to customer questions was: *I don't know. Let's put a breakpoint somewhere*!

Most of the time of my visit was spent debugging. We located misbehaving API source code, placed a breakpoint somewhere and run the whole application. After few steps over we usually know what the problem is and could either workaround it, or modify the NetBeans source code to better suite customer needs as I am not afraid to enhance API compatibly. At the end the customer was satisfied - but was it really necessary for me to be there? Could not the guys use the debugger themselves?

When my visit was over, I ironically commented that I did nothing else than debugging and patching NetBeans. At least I was useful to have the rights to make sure the patches get included in next version, I concluded. The customer's reply was shocking: *No, you are more valuable for knowing where to place a breakpoint!*

Does the ability to place a breakpoint becomes the most important skill for a modern developer?

**When Things Don't Happen**

Debuggers were designed to allow developers to step through their own code. It was certainly a huge step forward when such feature hit market for the first time. The ability to understand the behavior of your program by stepping through line by line and seeing the state of memory and variables is certainly useful. It is much easier than running the whole program and then trying to guess on which line something went wrong.

However the proliferation of open source libraries and the fact that these days we rather assemble applications than write them makes us use the debugger on foreign code rather than our own. Moreover we are no longer using just simple libraries (where we just made calls to them and continued our execution when the call returned), we are using frameworks. As a result it is the framework that makes calls to us depending on some declarative registration. Debugging our own code is still the same, however when one needs to find out why something did not happen - why the framework decided not to call us - we may get into tougher situation. Are our debuggers good enough to help us solve such problems?

Not really. Debuggers were originally designed to help us fix our own code when it misbehaved. Using them to find out why things don't happen is not easy.

**Side by Side Debugging**

Few years ago I was working on a project to bring JDeveloper and NetBeans closer to each other. We started by replacing JDeveloper's module system based on Equinox by Netbinox (a mixture of Equinox and NetBeans that was the fastest OSGi container on the planet at that time). You can imagine replacing something as fundamental as a base module system involves a lot of debugging.

The best approach was to run both systems side by side and watch for the differences. For a certain period of time we had a possibility to run the JDeveloper either on top of Equinox or on top of Netbinox - just by passing in different system property. When something went wrong under Netbinox, I performed the same operation in Equinox, observed the behavior by stepping

through the affected method line by line and then compared the behavior with Netbinox. Usually it was quite obvious where the difference is and yes, there were few.

This kind of side by side debugging is useful for finding out what is wrong when things don't happen. Because we had the old Equinox version that supposedly worked fine one could put a breakpoint to the place that should be executed, when the breakpoint is hit in the old version, just remember the stack. The problem in the new Netbinox version had to be along the stack. Placing another breakpoint (or breakpoints - depends on luck) to methods higher on the stack must once reveal a place where the code behaves the same in both versions. Then one could again switch to line by line debugging and find out where the difference between Equinox and Netbinox is.

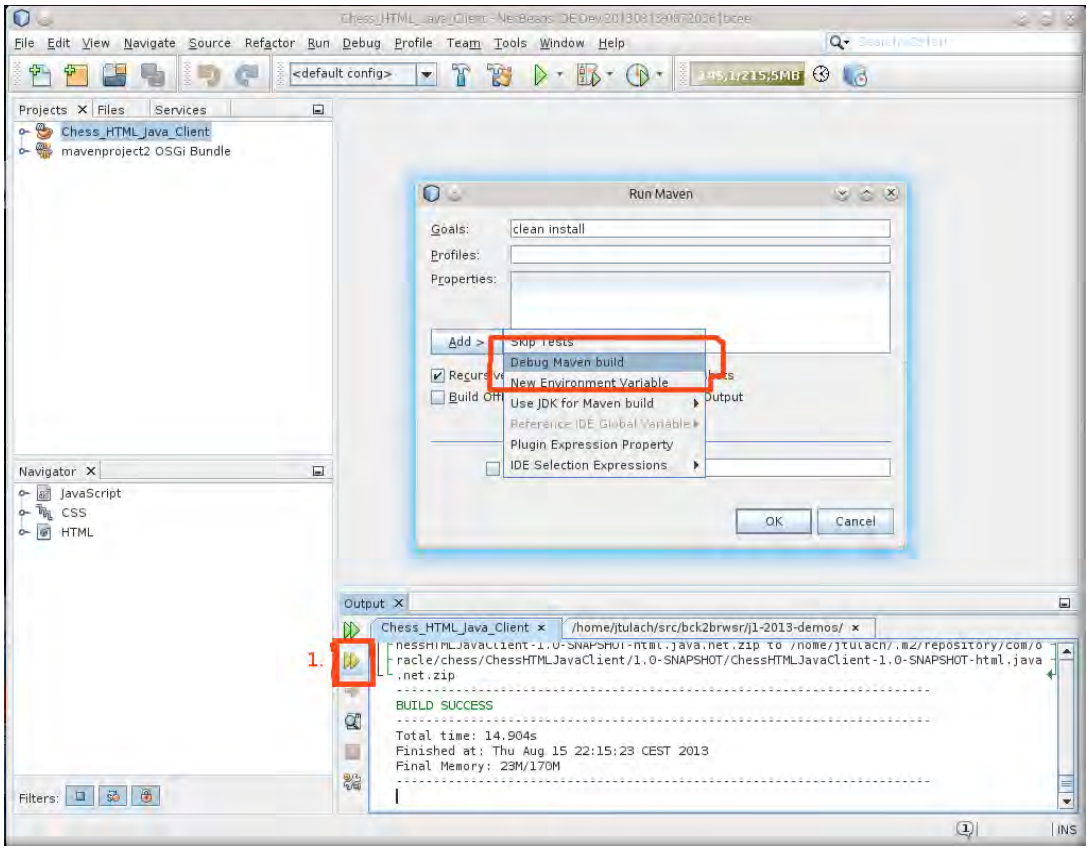
Side by side debugging is not useful only when we are writing a new version of a system. Even if something is wrong when we are using a framework, we can usually find a tutorial or demo application that is supposed to perform similar thing as our application. At that moment, we can compare the behavior of the working demo with the misbehavior of our application. Side by side debugging is useful for everybody.

**Insufficient Documentation of Maven Tasks**

Do you know the difference between Ant and Maven? In Ant you described what to do and make calls to individual Ant tasks. On the other hand in Maven you describe what should be done and let the system do it for you. That is more declarative, but also more framework like: often it is hard to find out why some actions are not performed the way you'd like them to be.

Recently we were writing a Maven plugin in Ant (yes, that is possible, both projects are supported by Apache foundation and as such they seek some synergy). Documentation how to achieve this is available, however it covers only the most common usecases. When you need to take it to extreme, use not only simple Ant script, but rather a set of files that work in harmony, you need to understand what is happening. The only way to do it is to use debugger. Can you debug your build?

The good thing about Maven and its NetBeans IDE integration is that debugging your build is easier than ever. Just execute your build (by invoking any goal on your project), let it fail and then restart it in debug mode. Press the yellow arrow in the output window and when a dialog pops up, select debug maven build. See the picture below for visual explanation.




Knowing how to debug your system is an essential knowledge that allows you to worship selective cluelessness. This applies not only to Maven, but to any system one happens to work with.

**Debugging without Source Code**

Debugger can be useful even for understanding a system without a source code. It is just essential to know where to place a breakpoint. Once I was trying to understand the behavior of iOS's WebUI which of course is not open source, but luckily Gdb (as well as many other debuggers) can place a breakpoint to a method and stop the execution when the method is called. As we needed to find out why the WebUI does not load a resource using our URL, we added a breakpoint to constructor of NURL. As soon as the breakpoint was hit, we could inspect the stack, see names of methods and even values of registers. Based on that we could get a clue of what was going on.

Once I was inspecting a closed source Java implementation of a Mylyn connector API. The system had three layers: User interface and connector API was open source, the library in middle was not. Placing breakpoints into the bottom connector API, I obtained very good understanding of what the blackbox in middle does. It is all about knowing where to place the breakpoints.

Have your Java application exited without a reason? Try to place breakpoint to System.exit(). If that does not work, create an exception breakpoint to stop when an exception is thrown.

Trying to influence behavior of your application with a system property and it does not work? Place a conditional breakpoint to Properties.getProperty(name) to stop when name is equal to name of your property.

Curious to know why a communication to an HTTP server fails? Place a breakpoint to URL constructor. Etc.

**Use the Debugger. Stupid!**

I am glad when people use NetBeans Platform, however it drives me mad when they approach me with a question that could easily be answered by debugging. In such situation I am tempted to say: *Use the debugger stupid!*. Usually I choose different words, but in case you see me to close a Bugzilla issue, or reply to an email with a reference to debugger, you know what you should do: *Use the debugger, stupid!*

**References**

[1] http://www.netbeans.org/

[2] Practical API Design http://practical.apidesign.org/

[3] 20 API Paradoxes http://buy.apidesign.org/

## What is the Mikado Method?

Ola Ellnestam, Daniel Brolund, http://www.manning.com/ellnestam/

This article is based on the book The Mikado Method, to be published in Spring 2014. This eBook is available through the Manning Early Access Program (MEAP). Download the eBook instantly from manning.com. All print book purchases include free digital formats (PDF, ePub and Kindle). Visit the book's page for more information based on The Mikado Method. This content is being reproduced here by permission from Manning Publications

How many times have you wanted to fix something, and while doing it not break the code? Or think of all the times that development work hasn't started from an empty codebase and you've inherited the constraints of the previous team. What you'd normally do is look over the documentation left behind; if there are automatic tests you'd see if they pass. Well, what happens if there aren't any tests left behind, and all that's left is the source code? How do you understand what's going on without breaking the code? In this article, based on chapter 1 of The Mikado Method, the authors talk about how the Mikado Method is a structured way to make significant changes to complex code.

The Mikado Method is a structured way to make major changes to complex code. When a codebase gets larger and complicated, as they often do, there usually comes a time when you want to improve portions of it to meet new functional requirements, new legal requirements, or a new business model. You may also just want to change it to make it more comprehensible.

For small changes you can keep things in your head, but for larger ones the chances of getting lost in a jungle of dependencies or in a sea of broken code increases dramatically. The Mikado Method helps you visualize, plan, and perform business value-focused improvements over several iterations and increments of work, without ever having a broken codebase during the process.

The framework that the Method provides can help individuals and whole teams to morph a system into a new desired shape. The Method itself is straightforward and simple and can be used by anyone at any time. In this article, we are going to look at the core concepts, benefits, and when you can use it.

### Basic concepts

There are four basic and well-known concepts that summarize the "process" of the Mikado Method: Set a goal, Experiments, Visualization, and Undo. These concepts, when used together in the Mikado context, create the core of the method itself.

Without these key pieces, the method wouldn't be able to help you make changes without breaking the codebase. By no means are these concepts new, but put together they become very powerful. In the context of the method they may serve a different purpose than how you might know them.

### Set a goal

To set a goal, think about what you want for the future. For instance, if you have a package with several web services that need to grow but it is already responsible for too much, perhaps what you'd want for the goal to be is "to have the admin services be in a separate package that can be deployed without the customer web services."

After you clearly state the goal, write it down. The goal serves two purposes: 1) represents a starting point for the change 2) determines if the method has achieved success or not.

**Experiments**

An experiment is a procedure that makes a discovery or establishes the validity of a hypothesis. In the context of the Mikado Method you use experiments to change the code so that you can see what parts of the system that breaks. Whatever is breaking gives you feedback on what types of prerequisites are needed before you can actually do that change. A typical experiment could be to move a method from one class to another, extract a class or reduce the scope of a variable. Those prerequisites are what you visualize.

**Visualization**

Visualization is when we write down the goal and the prerequisites to that goal.


Figure 1 A Mikado Graph with a goal and two prerequisites.

The picture shows a small map and the content of such maps normally comes after we've experimented and are ready to create what we call a Mikado Graph. Beside the changes to your codebase, the map is the only artifact of the Mikado Method. A refactoring map is the goal plus all the prerequisites of that goal, and it tells us what our next step is.

**Undo**

When an experiment for implementing a goal or a prerequisite has broken your system, and you have visualized what you need to change in the system to avoid it breaking, you want to your changes to restore a previously undo working state. In the Mikado method, you'll always visualize your prerequisites, and then undo your breaking changes. This process, experiments, visualization, and undo, is iterated for each of the prerequisites. In order for the experiments to be meaningful, the code needs to be in a known working state when the experiments start. If this isn't making sense to you now, it will when we get to the recipe where we take you step by step through the method.

Of these four concepts, the undo part is what people struggle with the most, because at first, undoing feels very unintuitive and wasteful. It's not waste, it is an important part of the learning.

**When to use the Mikado Method**

Figure 2 Being able to Change the shape of things is a highly desired skill

If we want to be successful software developers, we need to learn how to morph an existing system into a desired new shape.

Maybe you have tried to implement a new feature, but the system is constantly working against you? Maybe you've thought once or twice that it's time to stop developing new features for a while and clean up a bit? Maybe you've done a refactoring project, or you have tried to do a bigger improvement to your system, but wasn't able to pull it off so you just threw it all away?

We bet that you've been in at least one of the situations described above, and we know that the Mikado Method and this book could have helped. It doesn't really matter if the code was yours or someone else's; it doesn't matter if the code was old or new. Sooner or later that shiny new greenfield project where everything can fit in your head and changes are easy to perform will become more and more complex. As time passes the code fades just like grass does when it's heavily used and visited. The green grass field turns into a brown field, and sooner or later you, or your successors, become afraid of changing code. So, let's face it, we're stuck with brownfield development and we need to be able to morph code we're afraid of touching, in mid-flight. Let's look at a few common scenarios where the Mikado Method can help.

**Improve a system architecture on the fly**

When you've hit a wall and a design doesn't lend itself easily to change developers get frustrated. It could be an API that is hard to understand and your customers are complaining or your nightly batch jobs barely make it because the data that needs to be processed has increased by 10 times. It can be times like that when the code seems so complex and the only way to solve your problems is by stopping development and focus solely on improving the codebase for awhile or maybe run an improvement effort as a side project. Improvement projects make stakeholders nervous and rightfully so, because they see nothing of value coming out. The Mikado Method helps in changing the architecture in small steps, allowing improvements and continuous delivery of new features to co-exist, in the same branch.

**Brownfield development**

Brownfield development is probably the most common situation developers are in and in order to continue business upgrading and improving an existing application infrastructure is necessary. Whether it's adding a new feature, or altering functionality, the Mikado Method helps in these situations because it works with what you've got and improves upon it. Just like any other codebase, these also needs to change for several reasons, but often you don't know the whole code base inside out so changes become inefficient, or down right scary. The Mikado Method provides a way to take on a reasonable amount of improvements for each feature.

**Refactoring projects**

Imagine that you want to extract a reusable module in a heavily entangled system, or to replace an external API that has leaked deep into your codebase. Ideas and improvements like that are really big and they usually take several weeks, or even months, to complete. Making improvements for large tasks require a nondestructive way forward. The Mikado Method helps you uncover that nondestructive way and keeps you on track even if the effort takes months. This way refactoring projects can be avoided entirely.

## Benefits of the Method

We now know that the Mikado Method is a way to improve code more without breaking it, what situations we'd be in when we'd want to use it, now let's look at the benefits before we dive into how it works.

## Stability to the code base

Stakeholders will love the Mikado Method because it provides stability to the codebase while changing it. No more, “We can't release now, we're ironing out a few wrinkles.” The path to a change becomes a nondestructive one from lots of small changes instead of a big integration in the end. Due to its visual nature, interested parties can also follow along easily and watch the map evolve and then see how the changes are being performed and checked off on the map.

## Increases communication and collaboration

From a teams perspective it works really well too. By communicating the change map collaboration becomes easier and a change effort can be spread across the team. This way the whole teams competencies, abilities, and existing knowledge can be leveraged and the workload can also be distributed throughout the team.

## Lightweight and goal focused

Last, but not least, the Mikado Method supports an individual by being quick to learn and easy to use. The Method has very little ceremony and consists of a lightweight process that requires almost no additional tools, just pen and paper or a whiteboard. In its simplicity, it still helps you keep your eye on the price. As a bonus, you can use the refactoring map you get from the process to assist you when you reflection over the work done and this improves learning.

## Summary

You saw that the idea behind the Mikado Method is to remove the least amount of obstacles at a time in order to achieve real results without breaking code. We discussed some of the benefits and when to use the Mikado Method.

# Zucchini – a Visual Testing Framework for iOS Applications

Vasily Mikhaylichenko, LxMx, http://vaskas.me

Zucchini is an open source visual functional testing framework for iOS applications based on Apple UIAutomation. It allows the usage of freeform text for interaction scenarios definition.

**Web Site:** http://www.zucchiniframework.org
**Source Code:** https://github.com/zucchini-src/zucchini
**Version tested:** 0.7.3
**System requirements:** Mac OS X 10.6, Xcode 4.2, Ruby 1.9.3, ImageMagick, CoffeeScript
**License & Pricing:** BSD, MIT licenses
**Support:** https://groups.google.com/d/forum/zucchini-discuss

## Introduction

iOS mobile platform is uniquely consistent in terms of the operating system versions distribution [1] - as well as the devices range. With new iPhones being released once a year iOS developers get plenty of time to adapt to the new devices which also means more time to spend on polishing the actual applications. With iPhones only comprising 13% [2] of the overall smartphone sales, Apple's AppStore still earns 2.6 times more revenue [3] than Google Play. Needless to say that iOS remains very appealing to developers.

With more than 900,000 applications [4] in the AppStore today, developers try hard to make their applications stand out from the crowd. A well-crafted graphical user interface is an important aspect to this. Of course, not only the application needs to look and feel good, it also needs to be reliable and crash-free.

As Steve Jobs said,

> *"Design is not just what it looks like and feels like. Design is how it works."*

## Testing iOS applications

A great way of ensuring that your application is working properly as you continue the development is to maintain a comprehensive test suite. This normally involves at least two levels with Unit tests going hand to hand with the codebase and functional tests being at the higher (often business domain) level.

When it comes to iOS, the first category is represented with such tools as Apple's own XCTest [5], a popular Kiwi [6] framework, Specta [7] (all three based on the OCUnit framework), as well as Pivotal's Cedar [8] framework. As exciting as this category is, it is not going to be the focus of the current article.

## Functional testing frameworks

Some of the popular functional testing frameworks for iOS applications include Apple's UIAutomation [9], KIF [10], MonkeyTalk [11], Calabash [12], Frank [13], Appium [14] and Zucchini, all quite different in terms of features and implementation. Below is a brief comparison between them.

| Name | Scenario language | Step definition language | Assertion target | Communication protocol | Requires Xcode project modification |
|---|---|---|---|---|---|
| KIF | Objective-C | – | Accessibility labels | Same codebase as the application | Yes |
| Frank | Cucumber | Ruby | UISpec selectors | HTTP to a server embedded into the app | Yes |
| Calabash | Cucumber | Ruby | UISpec selectors | HTTP to a server embedded into the app | Yes |
| MonkeyTalk | Own DSL or Javascript | – | Own object model | UIAutomation | Yes |
| Appium | Any language with a Selenium WebDriver library | | UIA elements | Selenium iOS driver | No |
| UIAutomation | Javascript | – | UIA elements | UIAutomation | No |
| Zucchini | Freeform | CoffeeScript | Screenshots and UIA elements | UIAutomation | No |

**Zucchini**

Zucchini is a functional testing framework for iOS with a set of distinct features:

- Usage of freeform text for interaction scenarios definition
- Assertions based on screenshots
- Usage of CoffeeScript for screen contents description

Zucchini is based on Apple's own UIAutomation solution which means that:

- The underlying test runner is continuously improved and supported by the vendor
- It uses a very short communication path to a device or the iOS Simulator through Apple's proprietary Instruments [15] software
- It doesn't involve any modifications to the source code of the tested application

The framework is also compatible with continuous integration tools like Jenkins through exit codes and TAP-compatible [16] output.

**Design**

The mission of Zucchini is to enable efficient testing of iOS applications which is easy both to get started with and to support as the project grows.

A typical Zucchini test scenario looks like this:

```
Start on the "Books" screen:
  Take a screenshot
  Tap "Add"

Then on the "New Book" screen:
  Clear the "Title" field
  Type "The Great Gatsby" in the "Title" field
  Type "F. Scott Fitzgerald" in the "Author" field
  Take a screenshot
  Confirm "Save"
```

To get it working, the application screens need to be backed up by CoffeeScript classes like this one:

```
class NewBookScreen extends Screen
  anchor: -> $("navigationBar[name=New Book]")

  constructor: ->
    super 'new-book'

    extend @elements,
    'Title' : -> view.tableViews()[0].cells()[0],
    'Author': -> view.tableViews()[0].cells()[1]
```

It's valid to say that the framework is targeted at developers and test automation engineers familiar with the basics of JavaScript or CoffeeScript.

Zucchini gives its users the access to UIAutomation APIs which means that Apple's official documentation can be fully leveraged while working with the framework.

**Workflow**

Zucchini is different from the other iOS testing frameworks in terms of the assertions that it enables. Instead of forcing the user to drill through the UI elements hierarchy to retrieve a label or textual contents of a particular element, it performs a visual comparison of screens against reference screenshots.

**Why test against screenshots?**

As I mentioned it at the start of the article, iOS as a platform is very consistent. Which means that there aren't that many devices to test on – and not so much difference in terms of visual representation of the same application on different devices.

Screenshots comparison enables testing of many things like:

- GUI elements alignment
- Visual content that can't be represented with text like graphics and photos
- Font faces

As you may think tests like these can be quite picky. And from our experience, this has been very useful.

This said, particular screen areas can still be masked away from the comparison.

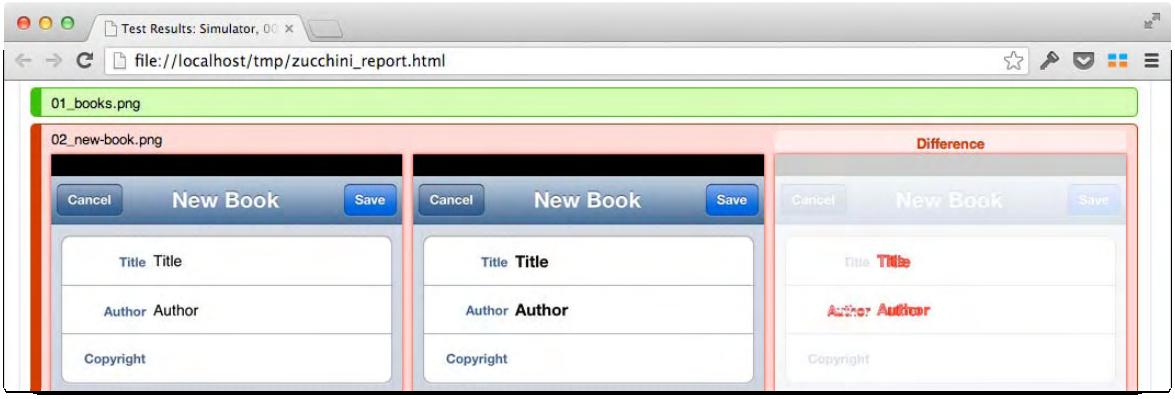



Figure 1. Zucchini spots a difference between a bold and a normal font

A good Zucchini test suite is meant to give you a complete visual representation of your application.

**Getting started**

Zucchini is easy to get installed [17] provided the existing Xcode installation.

You can then start by generating a project scaffold:
zucchini generate --project /path/to/my_project

As well as a feature scaffold for your first feature:
zucchini generate --feature /path/to/my_project/features/my_feature

Another good starting point is the Zucchini demo [18] which includes an easy to follow setup.

**Key concepts**

**Feature file**

Zucchini feature files describe user interaction scenarios leading to a particular state in the application. Once the state is achieved it's a good idea to take a screenshot to verify it against the expectation.

Feature files have a *.zucchini extension and are written in a close-to-natural language. While conceptually similar to Cucumber's [19] Gherkin language, it doesn't involve starting each line with Given, When, Then or And.

```
Then on the "Welcome" screen:
  Take a screenshot
  Tap "Menu"

Continue on the "Menu" screen:
  Take a screenshot named 'menu-first-time'
  Swipe right
  Tap "About"
```

### Screen class

The first line of each feature section corresponds to a screen. Each screen in the tested application should be backed up by a corresponding CoffeeScript class. These classes serve two purposes:

- **Providing an anchor element**

This element is verified to make sure that the current screen is the expected one. It should be unique to the screen.

- **Definition of screen-specific elements and actions**

Each line in a feature file corresponds to an action. Zucchini comes with a few standard actions but sometimes you might need to define a custom one to use the UIAutomation API directly:

```
class AuthorScreen extends Screen
  anchor: -> $("navigationBar[name=Author]")

  constructor: ->
   super 'author'

   extend @elements,
   'Author': -> $("textfield").first()

   extend @actions,
   'Scroll to "([^"]*)"$': (name) ->
     view.tableViews()[0].scrollToElementWithName(name)
```

By default, Zucchini tries to resolve the elements you specify in your steps (as in `Tap "Menu"`) by their name. If this doesn't work, you'll need to provide an `@elements` entry with a finder function, like in the example above. Fortunately, Zucchini integrates the Mechanic.js [20] selector engine as the `$` variable which makes navigating the UIAutomation elements hierarchy easier.

### Zucchini test report

Since Zucchini asserts the identity of screenshots, you'll need a set of reference screenshots to test against.

As you run Zucchini, the screenshots it takes are collected to the `run_data` directory inside the current feature directory.

You can then either copy them manually or use a subcommand to update all of your reference screenshots with the newly taken ones:

```
zucchini approve features/my_feature
```

### Mask

Sometimes you'll need to exclude particular screen regions from being compared. These could be animated areas, keyboard layout switchers, etc. Zucchini lets you provide screen masks for these specific cases. It applies the masks to both newly taken screenshots and the reference ones before the comparison.

**How it works**

You can run an individual feature or the whole directory using the subcommand:
zucchini run my_project/features

Zucchini works by compiling all the CoffeeScript source files into a single UIAutomation-compatible JavaScript file which it executes on a device or on the iOS Simulator through the Instruments command line tool. As the script finishes, it uses ImageMagick to mask and to compare the screenshots.

If you choose not to take any screenshots in your feature, the test outcome will depend on whether the UIAutomation part finishes successfully. Thus, Zucchini can be used just as a convenience wrapper / test runner around UIAutomation and Instruments.

**Summary**

Zucchini is a powerful way of ensuring the visual consistency of an iOS application.

It's a great choice if:

- You are using custom UI controls in the application, or
- Your application has got a significant amount of non-textual content, or
- You just want to give your UIAutomation tests more structure

The framework is a good starting point in functional testing of iOS applications. Simple to get started with, it should also appeal to developers who considered functional testing before but saw it as something too difficult or time-consuming.

## References


1. Apple finally charts iOS fragmentation, and it puts Android to shame, http://venturebeat.com/2013/06/21/apple-fragmentation-what-fragmentation
2. Apple's iOS loses marketshare to Android, Windows Phone,http://articles.washingtonpost.com/2013-08-07/business/41155227_1_smartphone-market-ramon-llamas-new-iphone
3. Google Play's App Revenue Share Up 8% Since November, But Apple Still Leads,http://techcrunch.com/2013/05/29/google-plays-app-revenue-share-up-8-since-november-but-apple-still-leads/
4. Apple: 900,000 apps in the App Store,http://www.theverge.com/2013/6/10/4412918/apple-stats-update-wwdc-2013
5. XCTest, https://developer.apple.com/technologies/tools/
6. Kiwi, https://github.com/allending/Kiwi
7. Specta, https://github.com/specta/specta
8. Cedar, https://github.com/pivotal/cedar
9. UIAutomation, https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html
10. KIF, https://github.com/kif-framework/KIF
11. MonkeyTalk, http://www.gorillalogic.com/monkeytalk
12. Calabash, http://calaba.sh/
13. Frank, http://www.testingwithfrank.com/
14. Appium, http://appium.io/
15. Instruments, https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html
16. Test Anything Protocol, http://en.wikipedia.org/wiki/Test_Anything_Protocol
17. Zucchini Installation Notes, https://github.com/zucchini-src/zucchini/blob/master/README.md
18. Zucchini demo project, https://github.com/zucchini-src/zucchini-demo
19. Cucumber, http://cukes.info/
20. Mechanic, a CSS-style selector engine for iOS UIAutomation, https://github.com/jaykz52/mechanic

# Portofino - A Java Web Application Framework

Author: Carlo Cassinari carlo [at] celeweb.eu

Nowadays web applications are required both to be innovative and to interact with legacy systems. As usual, productivity and solid architectures are essential requirements.

Portofino allows you to develop web applications from existing databases, by connecting to them and automatically creating a series of CRUD pages. Then it allows you to improve the application by creating pages like in a CMS, using enterprise components defined out of the box, or developing new functionalities.

**Web Site**: http://www.manydesigns.com/en/portofino
**Download:** http://sourceforge.net/projects/portofino/
**Version Tested**: 4.0.10
**License & Pricing**: LGPL (GNU Lesser General Public License), Free
**Support**: Mailing list http://groups.google.com/group/manydesigns-portofino or commercial support http://www.manydesigns.com/en/portofino/support


Figure 1. An application developed with Portofino

Portofino is an open source web application framework written in Java. Its main purpose is to create a complete web application by connecting to an existing database. You use a web wizard, set the database connection, specify a few other options and the framework creates a basic application structure with CRUD pages on the tables. You, as a developer, will avoid wasting time to create pages for the common data access pages (e.g. search, creation and editing pages). But you can also create new tables, and, on the Portofino side, you just need to define a query to have new pages built.

This is just a starting point. Portofino can be used as a powerful CMS. The pages are organized in a hierarchical tree, so you add, delete, move or copy pages anywhere in the tree. Pages are of different types and can contain a single content or many contents visually arranged on the page.

You can make your website good looking and functional at the same time.

The operations so far introduced are all done via web in a RAD-like fashion.

Another purpose of Portofino is to increase the productivity and ease of development (typical of other languages such as PHP or Ruby), with the added benefit of using enterprise level libraries that are de facto standards. It uses Hibernate for the persistence [1], Stripes as the MVC framework [2], Apache Shiro for authentication and authorization [3]. Each page has an associated action written in Groovy that allows easy customization on a live system without the need to compile and redeploy.

In the next sections we will see how to start to develop a typical project in Portofino.

**Connecting to an existing database**

Download Portofino from SourceForge: http://sourceforge.net/projects/portofino. The package contains an Apache Tomcat with a Portofino webapp. Launch Tomcat and see Portofino running at http://localhost:8080. Portofino does not need a database to start and to be used as a CMS. It allows you to create pages with a WYSIWYG html editor and to upload resources like images or documents.

If you have an existing database (or more than one), then you can use the wizard and let Portofino create pages automatically. Launch the wizard procedure from the home page. Connect via JDBC and follow the four steps of the wizard.

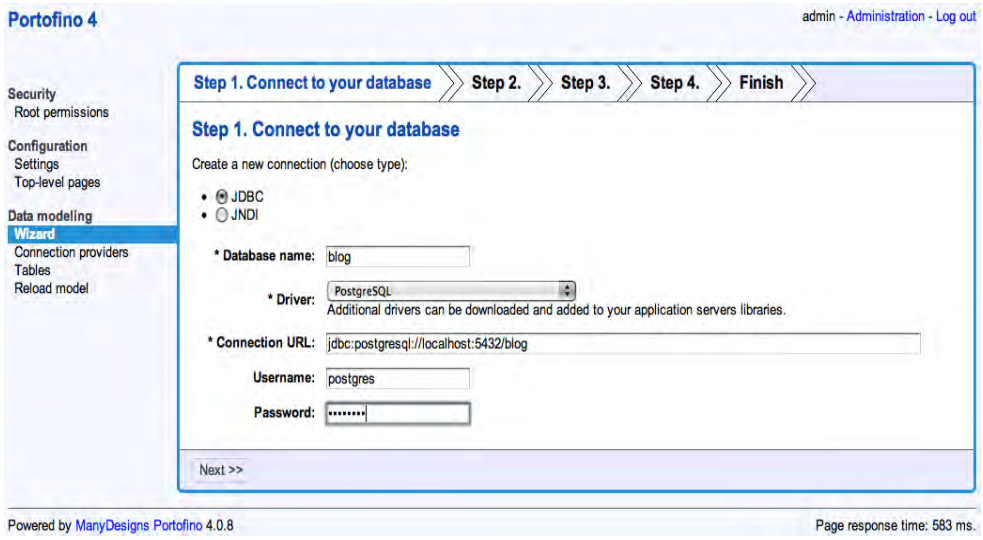



Figure 2. Connecting to an existing database

The wizard will ask you to select the schema and the tables of your database and allows you to select the users and groups tables, if you have them, to be used for authentication and authorization. At the end Portofino creates a series of CRUD pages, one for each table in the database.

These pages allow you to search, create, read, update and delete data from the associated table (defining a Hibernate query). Fields are searchable, results are paginated, columns are sortable, there are bulk edit and delete, autocomplete fields, fields validation. All these features are out of the box and do not require writing a single line of code. Furthermore, you can customize what fields to view, what will be searchable and the sorting. You can also remap field types instead of

the default taken from the database (e.g. an integer with values 0-1 can be mapped to a boolean). Productivity increases avoiding any unnecessary wastage of time in the development of common operations.

Analyzing foreign keys, Portofino organizes the pages hierarchically. Subordinate records will be positioned under their master record (e.g. “Customers” will be under “Companies”, or “Comments” will be under “Blog Posts” like in the following picture).

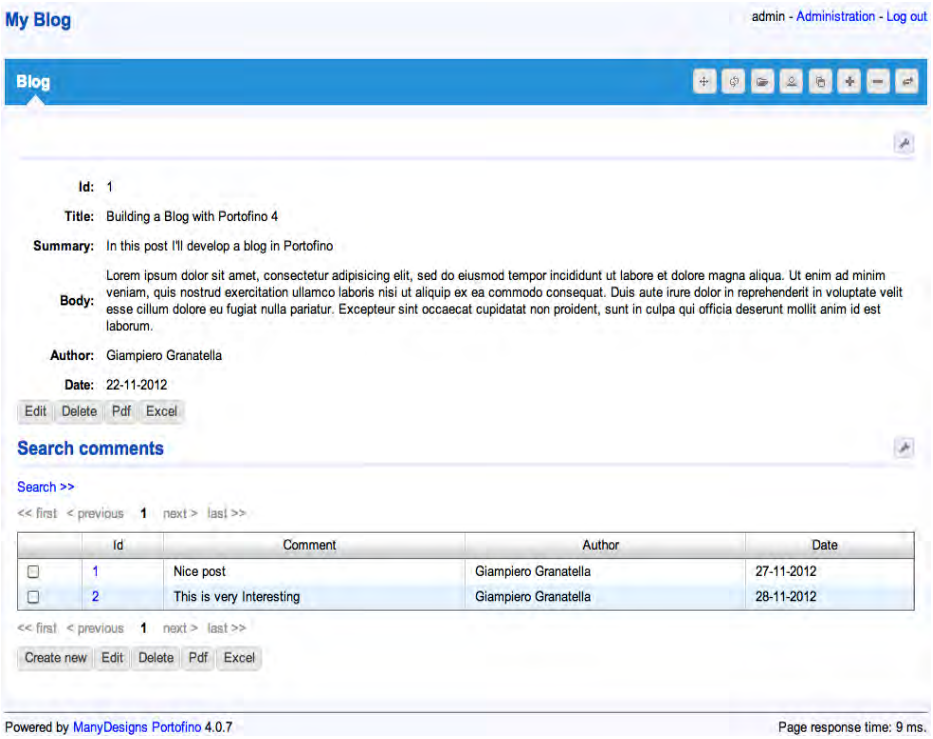



Figure 3. An example master-detail CRUD page

By default each page shows all the records but you can configure the generated pages and filter the data, you create a view for example with a “where” clause in the query.

You can also create connections to several databases, which allows you to use legacy data sources and new ones at the same time. Portofino allows virtual relationships between records in different databases. If you are building the database from scratch, Portofino supports Liquibase [4] for database versioning.

**Improving the page content and appearance**

The wizard is the first tool to use. It creates a working, although not yet complete application. When it finishes you can start working on the generated pages and create new ones (html pages, charts, ...).

This is a common refinement step. The pages, created by the wizard, are reorganized and new pages are created. You can create html pages as introductory text, summary or documentation.

Pages are organized hierarchically and you can see this structure in the left menu, which offers a navigation from the root page to the leafs. To further simplify navigation, each page shows breadcrumbs.

You can create text pages and attach them to the navigation tree. Page contents can be edited on-line. Simply visit a page, click on "edit", make the changes using the on line html editor. You can apply formatting, link other pages, embed images in the text or attach files to pages.

**Familiarizing with the application structure**

On the file system, the app is located under /apps/default which has the following directories:

- **blobs**, a directory with blobs (documents stored on the file system and available to pages);
- **dbs**, the Liquibase files that track database changesets and version;
- **groovy**, a common Groovy classpath;
- **pages**, all the pages organized in a directory structure that reflects the URL structure; each page has an associated Groovy action and a number of XML configuration files;
- **web,** custom html, jsp, css, images, javascript, and other web resources.

The main editing and administration operations are done via web, but you can work off-line on the app's files as well. A text editor is enough or you can use an IDE (e.g. Eclipse, NetBeans, IntelliJ Idea).

The app directory can be versioned using a VCS systems (CVS, SVN, Mercurial, Git...). This is a straightforward and a good practice.

**Improving the application**

Portofino offers a many features that you will discover and learn starting use it.

- Authentication and authorization

Portofino uses Apache Shiro for authentication and authorization. You can authenticate users using a table on the database, LDAP, OpenId or Facebook. Users belong to groups and authorizations are based on groups. Each page has a set of permissions (which can be improved or customized) that can be granted to a group. You can set the permissions via web directly on the page.

- The calendar page

Many components of Portofino let you show the data with a particular view. A calendar page shows events in a monthly or agenda view. Events may be taken from the database or from other sources. Any temporal data can be an event, e.g., the milestones of a project, the due date of a payment, the shipment date of a product. An application can have more calendars and events can be related to a user or a group of users.

- Emails

Portofino has an email queue that provides enhanced resilience to ensure that messages are not lost even during a temporary SMTP failure.

- Scheduled operations

Portofino integrates Quartz [5] so that you can write tasks and schedule them to be executed

- Chart page

A chart page uses the integrated JFreeChart library, with different chart types, tooltips, captions and colors. You can visualize the data with pie charts, bar charts, or other kinds and you can make the charts interactive with portions of the chart pointing to a customizable URL.

- Many to many page

This kind of page allows the user to check or uncheck from a list of options that is backed by a many-to-many relationship on the database.

- Custom action or definition of new kind of pages

Portofino is a flexible framework. You can create a “Custom page” that let you define a business logic, interact with the model, and finally redirect to a custom jsp to view the results. This is simple and can be done with a Groovy action (without recompiling or redeploying). At the same time, you can create new kinds of pages to be reused in your applications.

**Deeper customizations**

Each page has a dedicated groovy action. For example, a '/company' url has a dedicated /pages/company/action.groovy script on the file system. Portofino provides hooks for every page, which are standard methods exposed to allow customization such as business logic, validation, special actions or redirections. When you modify a groovy action, it is immediately reloaded in the webapp with a fast turnaround and a significant increment of the productivity.

Hooks are not the only method to customize an application. Using certain Java annotations annotations in the Groovy action, you can add new buttons and button handlers in the interface, or you can respond to AJAX calls.

If you have utility classes that you want to use across many actions, there is a Groovy classpath in the application where you can put them (the directory Groovy under the default app). Again, any modifications made to the groovy files are immediately picked up by the system.

You can edit the Groovy actions directly in Portofino, using a simple on-line editor. Basic syntax highlighting is provided. You can also use your favorite IDE and debug Groovy scripts on the live system.

**Conclusions**

Portofino is an innovative web framework that lets you preserve your database and at the same time to have a solid and modern web applications with CRUD, CMS and many other features.

The wizard let you start with a complete web application in minutes. The use of Java and Groovy allows you to develop and customize your application in a fast, simple and robust way.

If you wish to know more about Portofino, download it and try it. For more information, visit http://www.manydesigns.com/en/portofino. There you can find the documentation, several tutorials and videos, the Javadocs, and more.

**References**

1. Hibernate http://www.hibernate.org
2. Stripes http://www.stripesframework.org/
3. Apache Shiro http://shiro.apache.org/
4. Liquibase http://www.liquibase.org,
5. Quartz http://quartz-scheduler.org/

# TerraER - an Academic Tool for ER Modeling

Henrique Rocha, henrique.rocha [at] gmail.com, Ricardo Terra, rterrabh [at] gmail.com

TerraER is a free open-source learning tool designed to aid students in the creation of entity-relationship models. Our main goal is to provide students with a tool that reflects exactly the data modeling concepts learned in the classroom.

**Web Site:** http://www.terraer.com.br/index_en.html
**Version tested:** TerraER 2.02
**System requirements:** Java 1.5 or higher
**License & Pricing:** Open Source GNU Public License, Freeware.
**Support:** Documentation available at our website and issue tracker system at https://github.com/rterrabh/TerraER

## Introduction

Data modeling is one part of the database conceptual design process. The entity-relationship model (ER model) is a largely used conceptual model proposed by Chen [1]. It defines the entities and the relationships between these entities. The ER model is simple and easy to understand, making it intelligible to both database designers and end users [2, 3, 4].

Academic database courses still adopt ER models to teach conceptual design. Nevertheless, we noticed a lack of modeling tools for this purpose. In practice, most modeling tools support logical design, which is a detailed model the designer proceeds after the conceptual model is complete. In view of such circumstances, academics (students and professors) are forced to use logical design tools instead, such as DBDesigner (www.fabforce.net/dbdesigner4), ERWin (www.erwin.com), etc.

The use of existing logical design tools - rather than conceptual design ones - has two major issues: (i) they may confuse students who are learning about conceptual modeling; and (ii) they were not developed for academic purposes. To address these shortcomings, we designed TerraER, a free open-source learning tool designed to aid students in the creation of ER models. Our main goal is to provide students with a tool that reflects exactly the data modeling concepts learned in the classroom.

## Installation

TerraER is distributed in a single JAR file publicly available for download at our web site www.terraer.com.br/download_en.html. The JAR file is auto-contained, i.e., it can be placed in any folder, does not require the installation of additional libraries, and does not change operating system files (e.g., windows registry). In a nutshell, TerraER requires only a Java Runtime Environment (JRE) previously installed on the target computer (www.java.com/en/download/index.jsp).

## Main Features

TerraER 2.02 has the following relevant features:

- Free and open-source (GPL);
- Simple and small but complete application (around 5 MB);

- Cross-platform (only needs a JRE);
- Internationalization (English and Portuguese);
- Persist models to XML files (a cross-platform format);
- Clipboard transfer;
- Model printing; and
- Easy to learn and intuitive interface.

**User Interface & Feature description**

TerraER was initially designed for academic purposes, i.e., to help students and professors in the task of creating ER models. Therefore, we developed the graphical user interface to be practical, intelligible, and intuitive (i.e., easy to learn and use).

The tool is a free, open-source application under the GNU Public License. Thereupon, we encourage users to directly contribute to the TerraER project (https://github.com/rterrabh/TerraER). As an example, a B.Sc. student has contributed to the project by developing the internationalization for the English language.

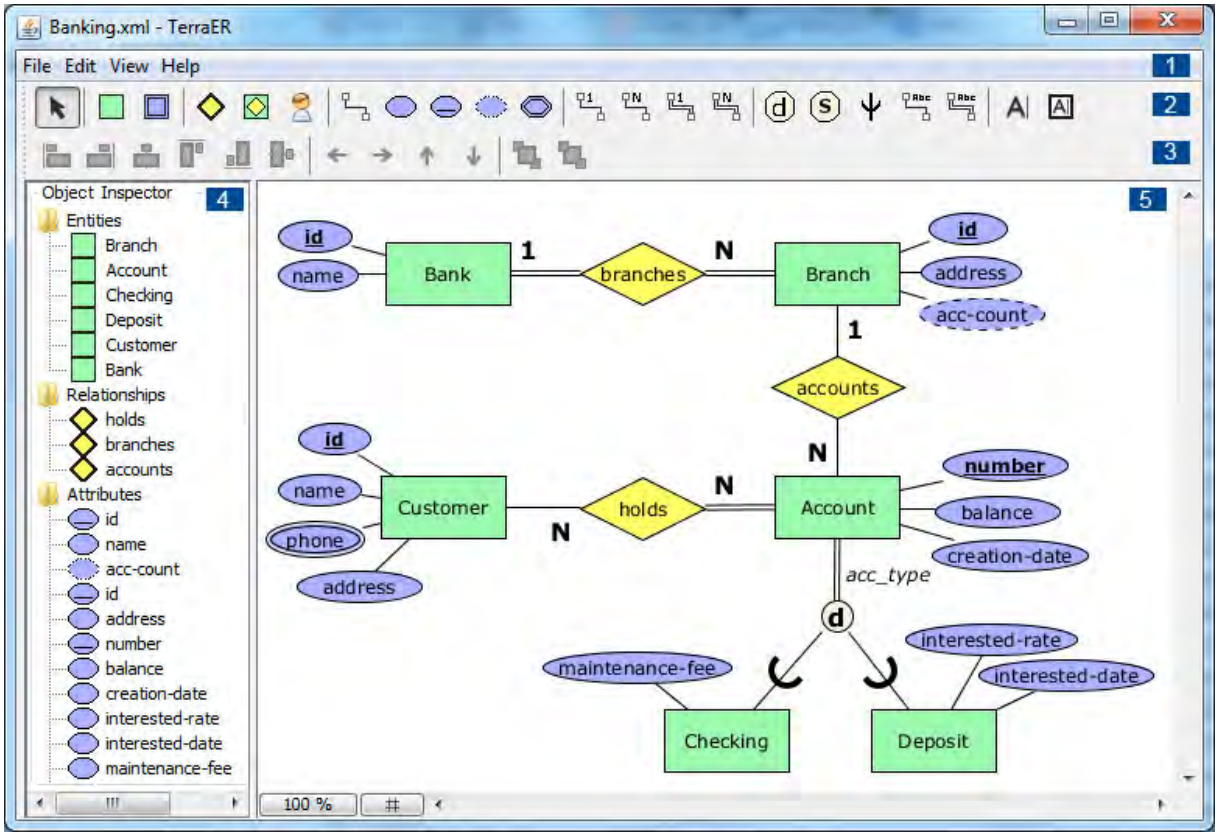



Figure 1 – Screenshot of TerraER 2.02 modeling a hypothetical Banking ER

The tool is developed in Java language, which makes TerraER a cross-platform application. In addition, new releases are always verified on Windows, Linux, and Mac OS.

Figure 1 illustrates the TerraER interface, which is organized into five main parts:

1. *Menu bar*: It provides the users with basic functionalities, such as saving, loading, and printing the models. More important, the models are saved in an XML-based format, which directly contributes to the cross-platform feature. In practice, models saved using an operating system can be loaded on a different one without any issue;
2. *Objects toolbar*: It provides commands to create ER model elements - such as entities, relationships, attributes, etc. - on *Chen*'s notation, as adopted by Elmasri and Navathe [3];
3. *Position toolbar:* It provides users with means to manipulate elements' position - such as alignment, overlapping, etc. - in order to support an elegant model design;
4. *Object inspector*: It lists the elements of the current ER model and allows user to select, remove, or edit them. In practice, this feature provides a quick and precise way to handle model's elements;
5. *Drawing area:* It shows the graphical view of the ER model under creation. The user may add and remove elements to the model. There is a *zoom* feature, which can be very useful when dealing with large models. Also, there is a *grid* feature that displays a grid to help users to position elements.

**Evaluation Experiment**

TerraER was initially designed as a teaching aid tool for academic database courses. In order to evaluate the applicability of our tool, we conducted a study in a Brazilian university to obtain the feedback from undergraduate students, which represent our target public.

In the study, we defined three assignments and divided the database course class in 10 groups. In the first assignment, five groups were required to use DBDesigner (a popular database design system) and the other five groups to use TerraER. In the second assignment, we swapped the groups. In the third and last assignment, each group could opt for which tool they prefer. In the last day of the class, the students filled an evaluation form.

The results clearly indicate a preference for TerraER as the model design tool. In fact, nine out of ten groups preferred TerraER to DBDesigner. According to the answers of the evaluation forms, the students argue that TerraER is easier to use and understand because it reflects the conceptual design learned inside the classroom.

**Supporting TerraER**

We welcome any help from the open-source community to support TerraER. For those not comfortable with their programming skills, it is possible to contribute with suggestions, donations, and even helping in the creation of a better documentation. TerraER source-code is available at https://github.com/rterrabh/TerraER, for those willing to play a more active roll.

**Conclusion**

Academic courses still adopt ER model for database modeling. However, there is a severe lack of tools designed for this particularly purpose. In view of such circumstances, academics (professors and students) do not have other option than to rely on tools that do not follow the concepts learned inside the class. Even though these are usually popular tools, they do not draw conceptual models but logical models instead. In practice, it may negatively impact the learning curve of the students.

To address this shortcoming, we developed TerraER with the specific purpose to be adopted in academic courses. Thereupon, we made the tool simple, easy to use, and complete w.r.t. ER modeling using *Chen*'s notation. More important, the study we conducted has shown students' preference for TerraER because it reflects the concepts learned in the classroom.

Last but not least, TerraER is a small, multi-platform, free, and open-source software system, which make the tool the proper choice for conceptual data modeling. As far as we can guarantee, TerraER is currently employed in databases courses in ten Brazilian universities.

**References**

[1] P. P. Chen. *The entity-relationship model – towards a unified view of data*. ACM Transactions on Database System, pages 9–36, 1976.

[2] S. Bagui and R. Earp. *Database Design Using Entity-Relationship Diagrams*. CRC Press LLC, 1964.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 6th edition, 2010.

[4] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 6th edition, 2010.

# UnQLite - an Embedded NoSQL Database

Mrad Chems Eddine, Symisc Systems, http://symisc.net/

UnQLite is an embedded NoSQL database engine. It's a standard Key/Value store similar to the more popular Berkeley DB and a document-store database similar to MongoDB with a built-in scripting language called Jx9 that looks like Javascript.

Unlike most other NoSQL databases, UnQLite does not have a separate server process. UnQLite reads and writes directly to ordinary disk files. A complete database with multiple collections is contained in a single disk file. The database file format is cross-platform, you can freely copy a database between 32-bit and 64-bit systems or between big-endian and little-endian architectures.

In this article, we shall introduce the concept behind UnQLite, it's architecture and a high level overview of its C/C++ API.

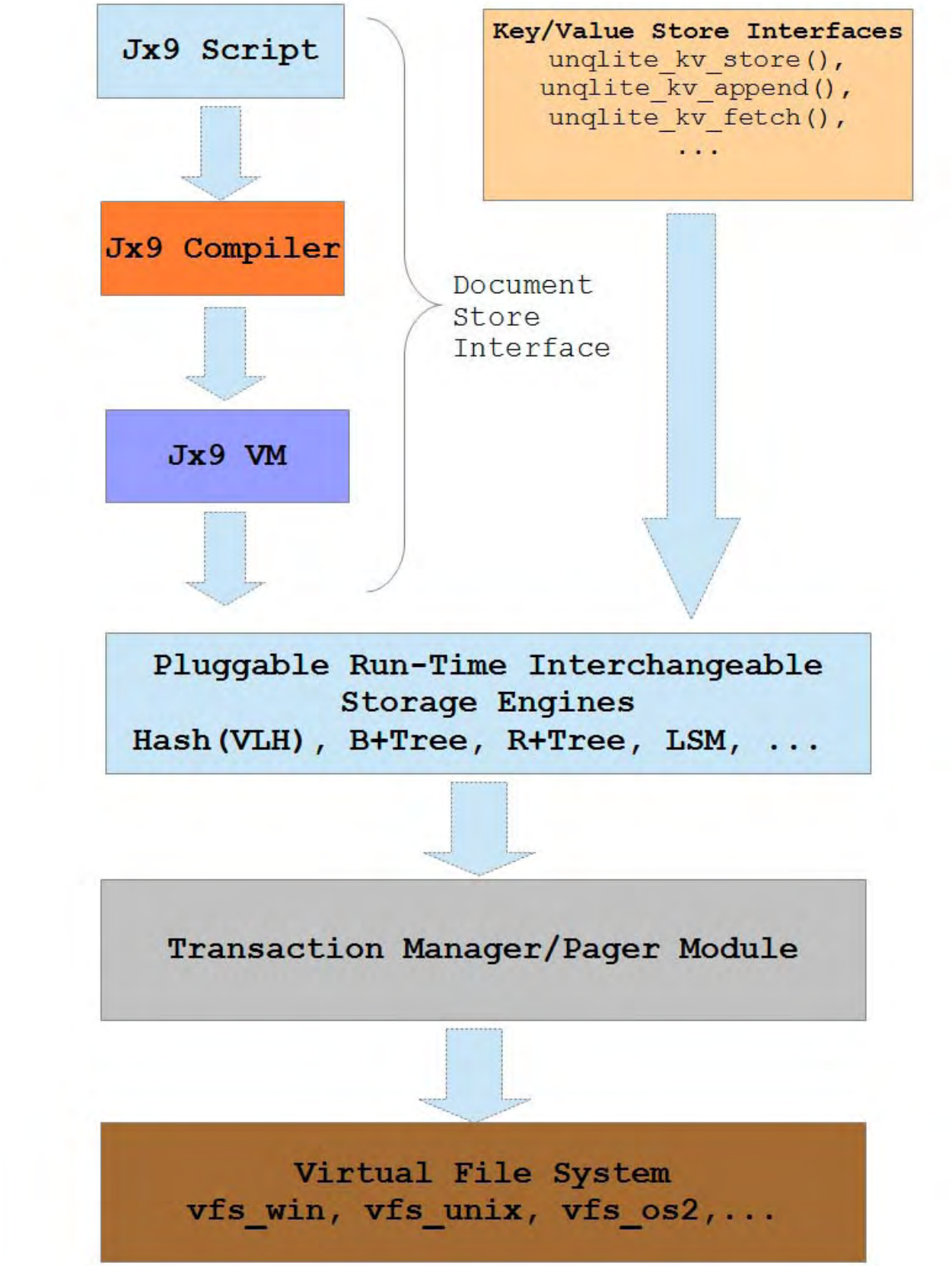
**Web Site**: http://unqlite.org
**Version tested**: 1.1.6
**System requirements**: UnQLite works on Windows and UNIX systems (Linux, FreeBSD, Oracle Solaris and Mac OS X)
**License & Pricing**: open source, http://unqlite.org/licensing.html
**Support**: http://unqlite.org/suport.html

**Development Timeline**

UnQLite was designed in the later 2012 when the Symisc (The company behind UnQLite) development team lead by Mrad Chems Eddine was working on a distributed P2P and Voip solutions similar in concepts to Skype and Kademlia. The challenge was to be able to store all nodes meta-information such as IP address, blobs, etc. locally and very efficiently.

The available solutions were not at our taste, Berkeley DB was too complex to embed in a plain C application and Google levelDB did not support transactions (Unlike UnQLite). After a moment of reflection, we decided to implement our own embedded database engine by taking the SQLite3 backend (i.e. the VFS layer, the locking mechanism and the Transaction manager) with our storage backend which should support on-disk as well in-memory operations plus the Jx9 stuff. A few months lather, UnQLite was stable enough and its performance rock!

**The UnQLite Architecture**



Like most modern database engines, UnQLite is built-up on layers, The upper-layers hence the document-store and the Key/Value store layers are presented to the host application via a set of exported interfaces (i.e. The UnQLite API).

The principal task of a database engine is to store and retrieve records. UnQLite support both structured and raw database record storage. The Document-store layer is used to store JSON docs (i.e. Objects, Arrays, Strings, etc.) in the database and is powered by the Jx9 programming language while the Key/Value store layer is used to store raw records in the database.

**Key/Value store layer.**

UnQLite is a standard key/value store similar to BerkeleyDB, Tokyo Cabinet, LevelDB, etc. but, with a rich feature set including support for transactions (ACID), concurrent reader, etc. Under the KV store, both keys and values are treated as simple arrays of bytes, so content can be anything from ASCII strings, binary blob and even disk files. The KV store layer is presented to host applications via a set of interfaces, these includes: unqlite_kv_store(), unqlite_kv_append(), unqlite_kv_fetch_callback(), etc.

**Document store layer.**

The document store that is used to store JSON docs (i.e. Objects, Arrays, Strings, etc.) in the database is powered by the Jx9 programming language.

Jx9 is an embeddable scripting language also called extension language designed to support general procedural programming with data description facilities. Jx9 is a Turing-Complete, dynamically typed programming language based on JSON and implemented as a library in the UnQLite core.

Jx9 is built with a ton of features and has a clean and familiar syntax similar to C and Javascript. Being an extension language, Jx9 has no notion of a *main* program, it only works embedded in a host application. The host program (UnQLite in our case) can write and read Jx9 variables and can register C/C++ functions to be called by Jx9 code.

**Pluggable Run-time Interchangeable Storage Engines.**

UnQLite works with run-time interchangeable storage engines (i.e. Hash, B+Tree, R+Tree, LSM, etc.). The storage engine works with key/value pairs where both the key and the value are byte arrays of arbitrary length and with no restrictions on content. UnQLite come with two built-in KV storage engine: A Virtual Linear Hash (VLH) storage engine is used for persistent on-disk databases with O(1) lookup time and an in-memory hash-table or Red-black tree storage engine is used for in-memory databases. Future versions of UnQLite might add other built-in storage engines (i.e. LSM).

**Transaction Manger/Pager module.**

The underlying storage engine requests information from the disk in fixed-size chunks. The default chunk size is 4096 bytes but can vary between 512 and 65536 bytes. The pager module is responsible for reading, writing, and caching these chunks. The pager module also provides the rollback and atomic commit abstraction and takes care of locking of the database file. The storage engine requests particular pages from the page cache and notifies the pager module when it wants to modify pages or commit or rollback changes. The pager module handles all the messy details of making sure the requests are handled quickly, safely, and efficiently.

**Virtual File System.**

UnQLite is designed to run on multitude of platforms. In order to provide portability between these platforms (i.e. Between POSIX and Win32/64 operating systems), UnQLite uses an abstraction layer to interface with the operating system. Each supported operating system has its own implementation.

An instance of the unqlite_vfs object defines the interface between the UnQLite core and the underlying operating system. The "vfs" in the name of the object stands for "Virtual File System".

**Introduction to the UnQLite C/C++ API**

Early versions of UnQLite were very easy to learn since they only supported 12 C/C++ interfaces. But as the UnQLite engine has grown in capability, new C/C++ interfaces have been added so that now there are over 95 distinct APIs. This can be overwhelming to a new programmer. Fortunately, most of the C/C++ interfaces in UnQLite are very specialized and never need to be used. Despite having so many entry points, the core API is still relatively simple and easy to code to. This article aims to provide all of the background information needed to easily understand how the UnQLite database engine works.

The principal task of a database engine is to store and retrieve records. UnQLite support both structured and raw database record storage.

In order to accomplish this purpose, the developer needs to know about two objects:

- The Database Engine Handle: unqlite
- The UnQLite (Via Jx9) Virtual Machine Object: unqlite_vm

The database engine handle and optionally, the virtual machine object are controlled by a small set of C/C++ interface routines. The dozens of the C/C++ interface routines and two objects form the core functionality of UnQLite. The developer who understands them will have a good foundation for using UnQLite.

**Typical Usage of the Core C/C++ Interfaces**

An application that wants to use UnQLite will typically use the following interfaces with their optional components such as the transaction manager and the cursors interfaces. Note that an application may switch between the Key/Value store and the Document-store interfaces without any problem.

| | |
|---|---|
| unqlite_open() | This routine opens a connection to an UnQLite database file and returns a database handle. This is often the first UnQLite API call that an application makes and is a prerequisite for most other UnQLite APIs. Many UnQLite interfaces require a pointer to the database handle as their first parameter and can be thought of as methods on the database handle. This routine is the constructor for the database handle. |

| | |
|---|---|
| unqlite_kv_store()<br>unqlite_kv_append()<br>unqlite_kv_store_fmt()<br>unqlite_kv_append_fmt()<br>unqlite_kv_delete()<br>unqlite_kv_fetch()<br>unqlite_kv_fetch_callback() | Key/Value Store Interfaces<br>Under the Key/Value store, both keys and values are treated as simple arrays of bytes, so content can anything from ASCII strings, binary blob and even disk files. This set of interfaces allows clients to store and retrieve raw database records efficiently regardless of the underlying KV storage engine. |
| unqlite_compile()<br>unqlite_compile_file()<br>unqlite_vm_config()<br>unqlite_vm_exec()<br>unqlite_vm_reset()<br>unqlite_vm_extract_variable()<br>unqlite_vm_release() | Document Store Interfaces<br>These set of interfaces works with the unqlite_vm object which is obtained after successful compilation of the target Jx9 script. Jx9 is the scripting language which power the document-store interface to UnQLite.<br><br>The Document-Store interface to UnQLite works as follows:<br>Obtain a new database handle via unqlite_open().<br>Compile your Jx9 script using one of the compile interfaces such as unqlite_compile() or unqlite_compile_file(). On successful compilation, the engine will automatically create an instance of this structure (unqlite_vm) and a pointer to this structure is made available to the caller.<br>When something goes wrong during compilation of the target Jx9 script due to a compile-time error, the caller must discard this pointer and fix its erroneous Jx9 code. Compile-time error logs can be extracted via a call to unqlite_config() with a configuration verb set to UNQLITE_CONFIG_JX9_ERR_LOG.<br>Optionally, configure the virtual machine using unqlite_vm_config().<br>Optionally, register one or more foreign functions or constants using the unqlite_create_function() or unqlite_create_constant() interfaces.<br>Execute the compiled Jx9 program by calling unqlite_vm_exec().<br>Optionally, extract the contents of one or more variables declared inside your Jx9 script using unqlite_vm_extract_variable().<br>Optionally, Reset the virtual machine using unqlite_vm_reset(), then go back to step 6. Do this zero or more times.<br>Destroy the virtual machine using unqlite_vm_release() |
| unqlite_kv_cursor_init()<br>unqlite_kv_cursor_first_entry()<br>unqlite_kv_cursor_seek()<br>unqlite_kv_cursor_last_entry()<br>*and many more* | Database Cursors<br>Cursors provide a mechanism by which you can iterate over the records in a database. Using cursors, you can seek, fetch, move, and delete database records. |
| unqlite_begin()<br>unqlite_commit()<br>unqlite_rollback() | Manual Transaction Manager<br>This set of interfaces allows the host application to manually start a write-transaction. Note that UnQLite is smart enough and will automatically start a write-transaction in the background when needed and so call to these routines is usually not necessary. |

| | |
|---|---|
| unqlite_close() | This routine is the destructor for the unqlite handle obtained by a prior successful call to unqlite_open().<br>Each database connection must be closed in order to avoid memory leaks and malformed database image. |

**Conclusion**

This article only mentions the basic UnQLite interfaces. The UnQLite library includes many other APIs implementing useful features that are not described here. A complete list of functions that form the UnQLite application programming interface is found at the C/C++ Interface Specification. Refer to that document for complete and authoritative information about all UnQLite interfaces.

# Karma - a Javascript Test Runner

Michael G Bielski, http://www.noinksoftware.com

Karma is a test runner for JavaScript that runs on Node.js [1]. It is very well suited to testing AngularJS [2] or any other JavaScript projects. You can use Karma to run tests with one of many popular JavaScript testing suites (Jasmine, Mocha, QUnit, etc.) and have those tests executed not only in the browsers of your choice, but also on the platform of your choice (desktop, phone, tablet.) Karma is highly configurable, integrates with popular continuous integration packages (Jenkins, Travis, and Semaphore) and has excellent plugin support.

**Website:** http://karma-runner.github.io
**Version Tested:** v0.10 running on NodeJS 0.8.19 running on Windows 7 Professional 64-bit
**License and Pricing:** Open Source
**Support:** User mailing list/group on Google Groups (https://groups.google.com/forum/?fromgroups#!forum/karma-users) and tagged questions on Stack Overflow (http://stackoverflow.com/questions/tagged/karma and http://stackoverflow.com/questions/tagged/karma-runner).

## Installation

The recommended installation location of Karma is in your global node_modules directory. Installing Karma is done via the Node Package Manager (NPM). From any command prompt, enter the command: npm install –g karma. Should you desire to install Karma to your local directory you can use: npm install karma –save-dev. This process will get you the default installation of Karma, which includes the karma-chrome-launcher, karma-coverage, and karma–jasmine plugins. There are also several plugins (a list can be found at https://npmjs.org/browse/keyword/karma-plugin) that you should consider installing to make your test management and output easier and more useful. I regularly use the karma–firefox-launcher and karma-ie-launcher plugins. All plugins should be installed before you try to configure or use Karma, but you can always add a new one later with no troubles at all. Installing plugins is just as easy as installing Karma. From any command prompt, enter the command: npm install -g-karma-<plugin name>.

## Configuration

Before you run Karma you must configure it. This is the most important step in setting up Karma. The easiest way to get started is to run the init command. In a command window, navigate to your project folder and enter karma init karma.conf.js (this is the default, but you can name the file whatever you want.) The questions are easy to answer and the net result is a properly structured Karma configuration file. You'll need to answer the following questions:

- What testing framework do you want to use?

  Jasmine, Mocha, and QUnit are installed by default and can be referenced by name, but if you have installed another one you should name it here. Because each framework is a plugin, you will need to list the plugin file in the plugins section of the configuration file as well (see the sample configuration file below for an example.)

- Do you want to use Require.js?

  Require.js is a lazy-loading framework that many developers use to minimize the initial script load times in the browser. If your project uses is, you need to answer YES to this question.

- Do you want to capture a browser automatically?

  You should respond with the browser name(s) that you want to use, one per line. Remember that you must have the matching –launcher plugin installed for each browser, and that only the karma–chrome-launcher and karma–chromeCanary-launcher plugin are installed by default. Entering a blank line will move you to the next question.

- What is the location of your source and test files?

  This path is relative to the location of Karma. You can enter an absolute path and be assured of directing Karma to the right location, or enter a relative path if your files are located below the Karma folder.

- Should any of the files included by the previous patterns be excluded?

  If you use a very broad pattern, you may want to exclude the folder where you store your images, or where there are no .js files to test. Tighter patterns take up more lines in the configuration file, but also eliminate the need for an exclude block.

- Do you want Karma to watch all the files and run tests on change?

  Having Karma run continuously is very helpful as you can see your tests and code evolve in a TDD environment. If you don't do TDD, you can opt to only run Karma when you are ready to execute your tests.

For the more adventurous or those that need more than just the basic configuration, you can edit the configuration file and tailor Karma to fit your needs. There are four main sections of the configuration file that you'll want to pay particular attention to: preprocessors, plugins, browsers, and files.

**Preprocessors**

Preprocessors enable you to do things to your code before it gets sent to the browser to have the tests run on it. Karma comes with the karma-coffee and karma-html2js preprocessors already installed. I regularly use the karma-coverage preprocessor to show me how much of my code I have covered with my tests. For a complete description of what any individual preprocessor does you should read the documentation on its website.

**Plugins**

While Karma comes with several plugins pre-installed, you need to tell which ones you want to use for running your tests. There are methods for dynamically determining which plugins get used, or you can just list them out. If your project doesn't change much with respect to how you are testing your code, the static listing is the easiest way to go.

**Browsers**

Browser support is critical for JavaScript. Karma comes with launchers pre-installed for Chrome, Chrome-Canary, and PhantomJS. Other launchers need to be installed if you want to test your JavaScript in other browsers. I regularly test in Chrome, IE, and Firefox. Karma takes care of capturing and killing the browser processes for you. This enables you to spend more time developing and less time doing the menial processes associated with testing.

The default time allowed to capture any one browser is one minute, which should be more than enough time, but you can change that value via the captureTimeout option in the configuration file. You will also need to tell Karma where the executable files are for all browsers except IE. If you want to capture your browser manually (from a tablet, for example) you can point the browser to http://<hostname>:<port> where <hostname> is the hostname or IP address of the computer where Karma is running, and <port> is the port you specified in the configuration file. By default this is port 9876, but if you have changed it in the port section of your configuration file then use that port.

**Files**

You need to tell Karma which files are required to run your project, which files contain your tests, and which files need to be tested. For convenience, in my configuration file I have broken these files into their respective groups, with required files first, test files second, and files to be tested last. The order in which they appear is the order that they will be included in the browser, so depending upon your project you may need to pay attention to it. Each file or group of files can be flagged for being watched by Karma, included in the browser <script> tag, and served by the Karma server. Files that are watched can trigger Karma to run all of the tests again, so pay particular attention to these. If you simply list all of your files, they will be watched, included, and served, which may result in performance loss during testing on older, slower computers.

Karma runs on NodeJS and the configuration is set up using module.exports so that it gets drawn into and consumed by NodeJS. Older versions of Karma did not have this feature, and they were slightly harder to configure correctly. A good sample of a Karma configuration file looks like this:

```
module.exports = function(config) {
  config.set({
        // base path, that will be used to resolve files and exclude
        basePath: '../..',

        frameworks: ['jasmine'],

        // list of files / patterns to load in the browser
        files: ['test/client/mocks.js', 'static/karma.src.js', 'test/client/*.spec.js'],

        // list of files to exclude
        exclude: [],

        // use dots reporter, as travis terminal does not support escaping sequences
        // possible values: 'dots', 'progress'
        reporters: ['progress', 'junit'],
```

```
// will be resolved to basePath (in the same way as files/exclude patterns) .....
junitReporter: {outputFile: 'test-results.xml'},

        // web server port
        port: 9876,

        // enable / disable colors in the output (reporters and logs)
        colors: true,

        // level of logging
        // possible values: config.LOG_DISABLE || config.LOG_ERROR || config.LOG_WARN || config.LOG_INFO || config.LOG_DEBUG
        logLevel: config.LOG_INFO,

        // enable / disable watching file and executing tests whenever any file changes
        autoWatch: true,

        // Start these browsers, currently available:
        // - Chrome, ChromeCanary, Firefox, Opera, Safari (only Mac), PhantomJS, IE (only Windows)
        browsers: [process.env.TRAVIS ? 'Firefox' : 'Chrome'],

        // If browser does not capture in given timeout [ms], kill it
        captureTimeout: 20000,

        // Auto run tests on start (when browsers are captured) and exit
        singleRun: false,

        // report which specs are slower than 500ms
        reportSlowerThan: 500,

        // compile coffee scripts
        preprocessors: {'**/*.coffee': 'coffee'},

        plugins: ['karma-jasmine','karma-chrome-launcher','karma-firefox-launcher','karma-junit-reporter']
  });
};
```

**Running Karma**

There are two main ways to run Karma: at the command line or via your IDE. I heavily favor running Karma from my IDE (WebStorm) because it keeps me on the same screen more. Regardless of which method you choose, running Karma does the following:

1. Starts a web server in Node.
2. Launches the specified browsers with a default URL that points to that web server.
3. Attempts to capture the browser session in each browser.
4. Upon capture of all browsers, the tests are run and the reports generated.

5. If you have opted to have Karma stay running and monitor files, each time one of those files changes the tests will be re-run and the reports re-generated.
6. If you have opted to only have Karma run once, each browser session is released and the browser is closed.

**Test Results**

Testing is only useful if you can see the results, and Karma has several options available. The progress reporter is inserted into the configuration file that the init process creates by default. This reporter gives you a complete listing of each test that is run, in the order that they are executed, and whether they pass or fail.

The dots reporter is included with Karma for those that run Travis. There are other reporters available as plugins (growl, Junit, Teamcity and Coverage) that can help make your testing process easier and more complete. Each reporter must be listed in the reporters section of the configuration file, and some reporters (such as the coverage reporter) require that they be referenced in the preprocessors section as well. The coverage reporter that is included with Karma is Istanbul [3] and I have found it very well suited for my needs and for use with Karma.

**IDE Integration**

Part of the beauty of Karma is how easily it can be integrated in to your IDE. Nearly all IDEs have a way to add an external tool that is launched on the command line. Configuring WebStorm 6 and older to run Karma as an external tool is covered in the introductory video that the creator (Vojta Jina) posted via YouTube [4] and can be found at the 7:39 mark in the video. There is a handy link to this part just below the video. There is also integration built into the upcoming WebStorm 7 (currently accessible via the Early Access Program.) It has been my experience that integrating with WebStorm produces an excellent development environment.

Support for configuring external tools isn't limited to JetBrains products. Visual Studio 2008 or newer in any flavor except the Express versions will allow you to configure external tools to launch from the menu (you would need to create a batch file with the requisite Karma commands in it and tell VS to launch that file.) Sublime is also a heavily favored text editor for web development, and it also supports external tools. As long as your favored IDE supports external tools in some way, you should be able to launch Karma and have it run tests.

**Documentation**

The vast majority of what you need to get up and running with Karma is contained in the Karma website. Most any other question can be answered by searching the Google Group or posting a question on Stack Overflow. Documentation for individual plugins will be found with those plugins. For anything that isn't covered, the Google Group is the best resource to try first, and Stack Overflow is a very close second.

## Actual Experience

A large number of developers don't test their JavaScript, and they really should. Too many of them think that testing is hard to set up and hard to do, and Karma is proof of the opposite. My team and I use Karma with Jasmine every day of the week. We found the installation and configuration to be fast and easy to follow. We went from downloading NodeJS to running tests via Karma in under an hour. When you consider that my team and I were totally new to testing, let alone testing JavaScript, this is pretty impressive. We chose Karma because it had support for AngularJS without any configuration changes, and because it would let us test our other scripts as well. In each of these areas it has excelled from day #1. We are now preparing to move our project to Yeoman and have no fear about how our tests will work because Yeoman fully supports Karma. We firmly believe that we could not have chosen a better tool to include in our daily routines.

## Summary

Karma is the preferred test runner for projects written with AngularJS and is well on its way to larger acceptance within the greater JavaScript community. Its plugin architecture makes it easily adaptable to other test suites and reporters, all of which add value to the core of Karma. In agile or continuous integration environments, Karma shines as an indispensable tool to development teams, providing an easy and reliable way to modify existing code and craft new code as part of TDD. It is rare that a day goes by without this tool running on my computer.

## References

[1] Node.js http://nodejs.org/

[2] AngularJS http://angularjs.org/

[3] Istanbul https://github.com/gotwarlost/istanbul

[4] Karma JavaScript Test Runner http://www.youtube.com/watch?v=MVw8N3hTfCI

## CodernityDB - a Fast Pure Python NoSQL Database

Jędrzej Nowak, Codernity, http://codernity.com

CodernityDB is an open source, pure Python without 3rd party dependency, fast, multi platform, schema-less, NoSQL database.

You can also call it a more advanced key-value database, with multiple key-values indexes (not an index that you probably know from SQL databases) in the same engine (for sure it's not "simple key/value store"). What do we mean by advanced key-value database? Imagine several "standard" key-value databases inside one database. You can also call it a database with one primary index and several secondary indexes. Having this layout and a programmable database behavior gives quite a lot of possibilities. Some of them will be described in this article. At first we will focus on very fast overview of CodernityDB architecture. A more detailed description will come in further sections.

**Web Site**: http://labs.codernity.com/codernitydb/
**Version described**: 0.4.2
**System requirements**: Python 2.6-2.7
**License & Pricing**: Apache 2.0
**Support**: directly via db [at] codernity.com and https://bitbucket.org/codernity/codernitydb

### General information

To install CodernityDB just run:

```
pip install CodernityDB
```

And that's all.

CodernityDB is fully written in Python 2.x. No compilation is needed at all. It's tested and compatible with:

- CPython 2.6, 2.7
- PyPy 1.6+ (and probably older)
- Jython 2.7a2+

It is mainly tested on Linux environments, but it will work everywhere where Python runs fine. You can find more details about test process in how it is tested in the documentation.

CodernityDB is one of projects developed and released by Codernity, so you can contact us directly in any case via e-mail. Please check the FAQ section first.

Do you want to contribute? Great! Then just fork our repository (https://bitbucket.org/codernity/codernitydb) on Bitbucket and do a pull request. It can't be easier! CodernityDB and all related projects are released under Apache 2.0 license. To fill a bug, please also use Bitbucket.

### CodernityDB index

What is this mysterious CodernityDB index?

At first, you have to know that there is one main index called `id`. CodernityDB object is kind off "smart wrapper" around different indexes. CodernityDB cannot work without the `id` index. It's the only requirement. If it's hard to you to understand, you can treat that mysterious `id`

index as key-value database (well, in fact it *is* a key/value store). Each index will index data by key and it will associate the value for it. When you insert data into the database, you **always** in fact insert it into the main index called `id`. Then the database passes your data to all other indexes. You can't insert directly into the index. The data inside the secondary indexes is associated with the primary one. So there is no need to duplicate stored data inside the index (`with_doc=True` when querying index). The exception from that rule is when you really care about performance, then having data also in secondary indexes doesn't require a background query to `id` index to get data from it. This is also the reason why you need to have an index in your database from the beginning, otherwise you will need to re-index your new or changed index, when you have records already in the database.

The CodernityDB index is nothing less or more than a python class that is added to the database. You can compare it to a read-only table that you may know from the SQL databases, or View from CouchDB. Here is an example of a very simple index that will index data by `x` value:

```
class XHashIndex(HashIndex):

    def __init__(self, *args, **kwargs):
        kwargs['key_format'] = 'I'
        super(XHashIndex, self).__init__(*args, **kwargs)

    def make_key_value(self, data):
        x = data.get['x']
        return x, None

    def make_key(self, key):
        return key
```

As you can see it is very easy python class. Nothing non standard for Pythonista, right ? Having such index in the database allows you to make queries about data in the database that has the "`x`" value. The important parts are `make_key_value` `make_key` and `key_format`. `make_key_value` function can return:

- `value, data:` data has to be a dictionary, record will be indexed
- `value, None:` no data associated with that record in this index (except main data from `id` index)
- `None:` record will be not indexed by this index

You can find a detailed description in the documentation. And don't worry, you will not need to add indexes every time you want to use the database. CodernityDB saves them on disk (in `_indexes` directory), and loads them when you open database.

Please look closer at the class definition: you will find there that our index is a subclass of `HashIndex`. In CodernityDB we have currently implemented:
* Hash Index - Hash map implementation
Pros

- Fast
- "Simple"

Cons

- Records are not in the order of insert / update / delete but in random order
- Can be queried only for given key, or iterate over all keys

* B+Tree Index (called also Tree index, because it's shorter) - B+Tree structure implementation
Pros
- Can be queried for range queries
- Records are in order (depending on your keys)

Cons
- Slower than Hash based indexes
- More "complicated" than Hash one

You should spend a while to decide which index is correct for your use case (or use cases). Currently you can define up to 255 indexes. Please keep in mind that having more indexes slows down the insert / update / delete operations, but it doesn't affect the get operations at all, because get is made directly from index.

You can perform following basic operations on indexes: * `get` - single get * `get_many` - get more records with the same key, key range * `all` - to get all records in given index. Because it is quite common to associate more than one record in secondary index with one record in primary we implemented something called: Multikey index.

Writing a whole python class just to change some parts of the method, like indexing `y` instead of `x`, would be not very user friendly. Thus we created something that we call IndexCreator. It is some kind of meta-language that allows you to create your simple indexes faster and much easier. Here is an example of exactly the same XHashIndex:

```
name = MyTestIndex
type = HashIndex
key_format = I
make_key_value:
x, None
```

When you add that index to CodernityDB, the custom logic behind IndexCreator creates the Python code from it. We created helpers for common things that you might need in your index. (IndexCreator docs), so once you get used to it, it is pretty straightforward.

As you maybe already noticed, you can split data to separate "tables/collections" with CodernityDB index (Tables, Collections in docs). All that you need is to have separate index for every table / collection that you want to have.

You should avoid operations directly on indexes. You should always run index methods / operations via database object like:

```
db.get('x', 15)
db.run('x', 'some_funct', *args, **kwargs)
```

**Usage**

You should have now a basic knowledge about CodernityDB internals and you may wonder how easy is to use CodernityDB. Here is an example::

```
#!/usr/bin/env python

from CodernityDB.database import Database

def main():
    db = Database('/tmp/tut1')
    db.create()
```

```
    for x in xrange(100):
        print db.insert(dict(x=x))

    for curr in db.all('id'):
        curr[x] += 1
        db.update(curr)

if __name__ == '__main__':
    main()
```

This is the fully working example. Adding this index from previous section will allow us to do for example:

```
print db.get('x', 15)
```

For detailed usage please refer to the quick tutorial on our web site.

**Index functions**

You can easily add index side functions to a database. While adding them to an embedded database might make no sense for you, adding them to the server version is very recommended. Those functions have direct access to the database and index objects. You can for example define a function for the `x` index :

```
def run_avg(self, db_obj, start, end):
    l = []
    gen = db_obj.get_many(
        'x', start=start, end=end, limit=-1, with_doc=True)
    for curr in gen:
        l.append(curr['doc']['t'])
    return sum(l) / len(l)
```

Then when you will execute it with:

```
db.run('x', 'avg', 0, 10)
```

You get the answer directly from the database. Please keep in mind, that changing the code of these functions doesn't require re-indexing your database.

**Server version**

CodernityDB is an embedded database engine by default, but we also created a server version. We created the CodernityDB-PyClient that allows you to use the server version without any code changes except:

```
from CodernityDBPyClient import client
client.setup()
```

Those 2 lines of will patch further CodernityDB imports to use the server version instead. You can migrate from the embedded version to the server version in seconds (+ time needed to download requirements from pypi). The Gevent library is strongly recommended for the server version

### Future of CodernityDB

We are currently working on or have already released (it depends when you will read this article) the following features:

- TCP server that comes with TCP client, exactly on the same way as HTTP one
- Permanent changes index, used for "simple" replication for example
- Message Queue system (single guaranteed delivery with confirmation)
- Change subscription / notification (subscribe to selected database events)

If these features are not yet released and you want to have them before the public release, send us a mail and tell what are you interested in.

### For advanced users

### ACID

CodernityDB never overwrites existing data. The id index is always consistent. Other indexes can be always restored, refreshed (CodernityDB.database.Database.reindex_index() operation) from it.

At a given time, just one writer is allowed to write into a single index (update / delete actions). Readers are never blocked. The write is first performed on storage, and then on index metadata. After every write operation, the index flushs the storage and metadata files. It means that in the worst case (power lost during write operation) the previous metadata and storage information will be valid. The database doesn't allow multiple object operations and has no support for typical transaction mechanism like SQL databases have.

Single object operation is fully atomic. To handle multiple updates to the same document we use the _rev (like CouchDB) field, which informs us about the document version. When rev is not matched with one from the database, the write operation is refused. There is also nothing like delayed write in the default CodernityDB implementation. After each write, internals and file buffers are flushed and then the write confirmation is returned to the user.

CodernityDB does not sync kernel buffers with the disk itself, it relies on system/kernel to do so. To be sure that the data is written to the disk please call fsync() or use CodernityDB.patch.patch_flush_fsync() to call fsync always when the flush is called after data modification.

### Sharding in indexes

If you expect that one of your database indexes (`id` is the most common one there) might be bigger than 4GB, you should shard it. It means that instead of having one big index, CodernityDB splits the index into parts, still leaving API as it would be single index. It gives you about 25% performance boost for free.

### Custom storage

If you need, you can define your custom storage. The default storage uses `marshal` to pack and unpack objects. It should be the best for most use cases, but you have to remember that you can serialize with it only basic types. Implementing a custom storage is very easy. For example you can implement a storage that uses `Pickle` (or `cPickle`) instead of `marshal`, then you will be able to store your custom classes and make some fancy object store. Anything is possible in fact. If you prefer you can implement a remote storage. The sky is the limit there. The other reason might be implementing a secure storage. You can define a storage like this and you will get an encrypted transparent storage mechanism. No one without access to the key will be able to decrypt it.

**Better Software & Agile Development Conference East** | November 10–15, 2013 | Sheraton Boston Hotel | Boston, MA

2 Conferences, 1 Registration, 1 Location! Register for one conference, perfect for any software professional, and receive full access to BOTH events! You get double the value, more content and subjects to choose from, and the freedom to attend both conferences at your convenience. 84 speakers, 24 Better Software Sessions, 24 Agile Development Sessions, 35 Tutorials, 4 Keynotes, a combined EXPO, and much more!Register by October 11, and you'll save up to $400 when you use code MTEM!

Go to http://www.sqe.com/go?MTBSCE13ED

---

**Agile Across the Board - Agile Business Conference London 9-10 October 2013** Exploring Agile as the mature and inclusive approach for the management, development and delivery of projects and products in any sector from any culture. There will be keynotes, interactive sessions, workshops and presentations, so come along and meet experts, learn from and share practical experiences and network with other professionals

Go to http://www.agileconference.org/

---

**Code Review Like A Boss!** Klocwork Cahoots is a flexible and easy-to-use code review tool that simplifies the review process by removing the overhead and complexity found in other tools. Designed for development teams of all sizes, Cahoots fits in the developer workflow to ensure code reviews are fast and effective.

Download a 30-day product trial today:
http://download.klocwork.com/products/cahoots?utm_source=mandt&utm_medium=ad&utm_campaign=enews-augsept

**METHODS & TOOLS** is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php