BAD ASS Documentation

Release 0.1.0

Julien Tayon

September 07, 2012

CONTENTS

Contents:

CHAPTER

ONE

BAD ASS

1.1 Backend Agnostic Development AsSynchronous Site

BAD ASS is just my collections of all the trends I see in the communities in which I feel I belong (web, python, Perl) that are strongly emerging and are currently killing the jumboframeworks.

It is all about coding less and achieving more

Obviously bad developers develops, good developer factorizes.

It covers as much as QA, architecture, design, coding strategies, coding convetions, conceptual integrity, protocols, security.

If you want a design pattern name so that you can toy with it (or choke on it) call it MVCeption : A MVC in a MVC.

1.2 You know I am BAD

BAD is the server side strategy. Backend agnostic refers in my head to the actual anti pattern I have seen so many time of people focusing on the storage facility (MySQL, CouchDB) and optimizing before even knowing their data.

> **Show me your code, and I shall be mystified, show me your data and I'll know everything there is to know**
> – Fred Broks

BAD is focusing on how to semantically present data in an intuitive way, without not too much complexity, while having a good security.

1.3 And my ASS is no chicken

ASS is about front side integration with the means of asynchronuous interfaces (HTML, Flash, GUI).

It also covers front to back communication and security best practices, and some tips on using the DOM as a database.

1.4 Finally we have got a badass here

By combining these two ideas, we can clearly see how to make front and back developpers cooperate easily. And how to be lazy.

CHAPTER

TWO

BAD NOT OVERFOCUSING ON THE STORAGE

Before even thinking about coding, know what you code.

2.1 The importance of a model

2.1.1 Conceptual Static Model

I have made countless technical interviews where CTO did not know their conceptual models but already knew they'd rely on stored procedure in their favourite RDBMS.

> Early optimization is the root of all evil.

Conception bugs are the most costly. And physical model are no replacement for conceptual models.

Developers have one craft with its jargon. They will mostly make an application for non developers with an established craft, with its own conventions. Conceptual Model is not only about abstraction, it is also about practical understanding, each terms should be defined exactly and if refering to technical, legal or accounting the norms should be explicitely quoted and made available so that developers can check the consistency when in doubt.

For instance if a database is about printing and their are pages and RV and costs. It is always a good idea to define what a page is, that a RV is a *recto verso*, and if there is a cost in an usual measure it is always nice to tell the unit and the precision needed.

2.1.2 Model Checklist

- is there an expert (or a consultant) in the room, if so wait until he leaves to begin working seriously (there is an high probability your so called expert is just an attention whore with self esteem issues) ;
- Cardinalities ;
- Exact definitions for all terms ;
- entity diagram ;
- prefere has_a, has_many over is_a (inheritance is evil, multiple inheritance is only good for a biological database of species taxonomia) ;
- volumetries ;
- units ;

- an estimation of the cost of transaction (some data maybe considered not vital like a comment on a post and making them transactionnal does not worths the trouble, others like a withdrawal on a bank account maybe considered as vital) ;
- are data polymorphic (a product should have a price, a description, and a serial number, but it may differs in additionnal data (colour, size, unit, picture, energy consumption) amongst different categories) ?
- the previous is synonym of is there a soft model ?
- do we use in our model the **exact** terms that are used contextually in the business we code for (price is not exact enough in ecommerce, it is usually a pricing without taxes included, and or discount) ?
- does it states any implementation ?

2.1.3 Model inspection

There is worst than making mistakes, it is doing nothing. If you are a doer, this first Conceptual Model might be full of stupid mistakes.

Well, let's live with it, people with ideas never commit themselves in the filthy mud of realisation, that's the reason they never fail.

First of all, if you have an implementation or a design specification, something is wrong, you are one step ahead in the physical model. It means you didn't not notice that you had an expert in your meeting, your work is ruined, you have to redo it from scratch.

Then it is time for pre-shooting bugs.

> Complexity is the ennemy of developpers.

It is time to see if there are ways of simplifying the model. First usual conceptual difficulties may arise from having being to good at your school and there are a few facts that should be remembered :

- short term memory range between 3 (stupid or tired people), to 7 (well rested autists and geniuses) items. If possible try to prune your depth of inclusion to a layer of 3 ;
- Garbage Collectors hate cyclic references. Try to have uncoupled data structures and try to cut all cyclic references, you are already getting rid of memory leaks and data corruption ;

Now that is done, let's fire in the hole :

- what is private to an entity, what is shared, what is public ? (begining the AAA)
- Most developers have a poor knowledge of database. Stick to the basics, ForeignKeys, usual data types, and try to limit the joints (most developpers will stupidly try to optmize them resulting in less than optimal results) ;
- Dont be too smart. What you think is trivial (like a table for nodes and recursive SQL request to buid a tree) may bot be for most developpers ;
- **Fucking check with an end user** that all your members name, units, definitions, are **exacts** ;
- **Check with an end user** that he understands your model.
- beware of exponential, when **n** items are connected to one another they have *n x (n - 1)* connections ;

  Complicated is easier to do than simple, and most people confuse an obscure design with a brillant one. Complex is not complicated : complex is having a lot of simple units interacting with each others. This is the definition of a complex system, hence something potentially chaotic.

2.1.4 Preparing the physical model

I did not speak of UML, or conventions, because, it is like style in code. You have to help people produce. Formalism is necessary, but it should come after having a model. You cannot ask people to focus on more than one objectives at a time. Conceptual model is about having the semantic, and the definitions exact !

> I don't think we should emphase on coding convention when people submit patches. It demotivates them, when they propose substential patches, and we can still help them afterward on the coding convention.
>
> – seen on python-dev

Now is the time for formalism. I share with you my KISS principles :

Convention are the way of lazyness, it is made to avoid guessing

KISS : How not to guess

- if a data in a structure is made for a joint use *tablename_id*
- **singular is simple** making plural out of a noun, is a more complex than it seems, if an attribute refers to more than one things, it returns a record. This is the context in Perl, duck typing in python, and ruby.
- CamelCase are less readable than under_score convention, plus some storage facilities dont discriminate between upper and lower cases ;
- no other langages than english in your name. Dont mix up languages because it makes your brain automatically switch in differents incompatible patterns of thinking ;
- Foreign Keys are Good ;
- Dont think an SQL dump of a database on your computer and freaking autogenerated documentation is a physical model. This is just crap.
- for every common data type : please use international standards (IETF, IEEE, ISO, MPEG ...) ;
- if you use international standard give the exact reference to the norm.

  s/([^@]+)@([.]+.[w])/ does not validate an email. only strings compliant with the RFC 5322 are.
- (local) Time and Dates, intervals are the biggest tar pits in CS. Be very careful for this datatype, and on the convention (if an application is worldwide, store time in UTC).
- internally use the ISO 8601 date format, since you sort lexicographically the date ;
- when you need a hash dont say you are gonna use **MD5**, this just proves you are a plain idiot (MD5 is broken, for many purpose, and has been for years). You need a hashing function.
- Use adapters to change from end user input format to internal representation;
- regexp validation is good since web is strings and it is portable between a lot of languages, and it stores very well in a cross plateform format known as JSON;
- if using regexp, dont dare recode your own home brewed regexps for Credit Card, IETF definitions, date. Use libraries unless you dont fear a psychopath developper know your personnal address.
- share your defintions across langage, and repositories. JSON is very handy;
- for cardinality dont use symbols, prefer sided arrows with words such as has_a, has_many, many_to_many, the brain interprets them faster, with less errors.
- double check orthographs ;
- check regionalisms stick with your conventional region (US or EN) (*color* for en_US *colour* for en_EN)
- check for typography ;

- check for inconsistencies in terms (start match with stop not end, statuses are no events) ;
- if using units, check unit integrity and stick to International System if you are free to choose ;
- Any developper/architect specifying or coding prices or any money related attributes in float deserve a slow painfull death. Decimal, Fixed Point tricks are the only acceptable way to have accurate price calculation. Rounding convention, precision **MUST** be specified ;
- Do diagramms have legends ? No, this is not a diagramm it is powerpoint engeneering, the same stuff that made challenger explode : it is worthless.
- Please make a representative example that non developper can grock so that they can make sure model is consistent.
- use unicode as an international representation for strings, utf8 encoding for any exchange (file, networks). (UTF8 is endianness unambiguous, unicode16 (32 if you are coding a chinese website with non simplified ideograms) is linear in performance for reaching offset n, or applying regexpes);
- endianness, CR/LF, IO are no simple problems.
- prepending _ to an attribute is quite a good hint to say this data is not to **"printed as is under any circonstances"** on the front end.
- dont follow any rules (like the aforementioned ones), they may not be convenient in your context, but above all **be consistent**.
- know if your storage is SQL or noSQL like, but unless a killer feature (Geographical entities) exists don't use any special features.
- **don't guess sizes when you can't** : it is legitim for a first name to be 2 letter long (being Li is legit), the only acceptable length for an email is the one described in the RFC ;
- If you need to store hyperlinked text, **never ever used HTML**. Mardown, Wiki, RST are made for this use, and they all have specialized WYSYWYG widgets.

2.1.5 Workflow / AAA

Here you normally have burnt around 1/7th and 1/10th of the whole effort. Your estimation is normally not given to the hierarchy but you know can estimate roughly how much time the project will actually take. And since you had an early interaction with the customer, you can apply the smoothness correction of intercommunication. If it is hard to have a contact with en end user now, it will be tougher later in the project, so add delays and stolen time in your estimations.

AAA = Authentication, Authorization, Accounting

This is one of the place where we have the most problems, especially from so called pros.

> **Science is the belief in the ignorance of the experts.** – Feynman (speaking of the teachers)

As stated on coding horror, we have all done the stupid mistakes of thinking we were smart when doing security. We all made the mistake of implementing the mechanism. **NO**. There are **AAA** frameworks, and dedicated **architecture**.

AAA as defined in RFC XXXX is a concept. It is the underlying concept between most security network mechanisms. By fitting in the concept, you'll be more likely having less difficulties at implementing them.

Authentication How is an entity identified, and allowed in a realm. How are granted the access, what are the session handled ? How to store enough information server side to identify a user? How to give the information to ensure an identity? How to handle access token ?

Authorization What defines an entity ? What are its profile (groups) and which permissions are related to the groups ?

Accounting What authenticated entity accessed what, when ? When did they accessed the system ?

Authentication

Hashing for instance is not **MD5**. MD4 is mathematically very close and faster. Plus MD5 algorithm has a non null collision probability, and there are image attacks disclosed for years on this. Using MD5 for doing your own crypting of private data (passwords for instance) is just the proof your are a moron that confuses the abstraction and the implementation.

This is not yet the time to think of authentication

Authorization

Well, this is important, since it will impact the authentication you can use.

KISS : try not to be intelligent, you are in the realm of hidden combinatories.

Entity for a B2C business, an entity should be a user. In B2B it should be a role linked to a person. In companies workers are fired and hired, and fonction stays. Base your model on role, not persons. One entity has one and only one role

Groups For small scale data, they may be used instead of permissions. For average models, these are sets of permissions related to a function (auditor, commiter, dispatcher). It can also be used to state the Organizational Unit the entity belongs to.

Permissions Be lazy, your data are probably tree like. A branch is like a directory, a leaf is like a file. Try to map a group to a model, and try to position your permissions in terms of read, write, execute.

Writing an admin backend is now just presenting a big matrix of group vs role where you check your 3 permissions for each case.

Data Integrity is also real security

The truth in online business is your wealth is your data. And they **MUST** be accurate enough. Enough is driven by the cost / price / risk assessment. Mostly, you need to read the contract signed with your customers to assess this. In doubt dont guess. You have a budget. Dont overdo it or you'll criple your interface speed. Don't underdo it or you may harm your most valuable asset : your reputation.

Security == complexity. Too much complexity result in slower interactions, this may be acceptable when wiring 50k$ but less likely when chatting with a gorgeous person you want to have social interactions with. It is a tradeoff.

Transactions

There is not transactions possible without a stateful connection be it enforced on the lowest layer, or simulated at application level. There is not transactions without a state transitions model.

The number of possible transitions increase exponentially in function of the available states. A brain cannot memorize 49 transitions or even 10 transitions.

If your model has objects made of other objects, you may want to rollback an incomplete object, unless the information can be safely stored for later modifications. This is the Dynamical Conceptual Model.

This is the time where dead locks are evaluated, locks are thoughts, automatics triggers for business rules positioned, User Interface scenarii (aka functional testing) written.

A functionnal test, is just a scenario that is tested. You will be probably test transitions from given states.

One of the most important scenario to describe is :

- user is not connected ;

- user tries to see private data ;
- error is triggered ;
- user authentifies ;
- user can read/write/execute legitimate private data ;
- ... (all revelants case taste ensuring data integrity)
- user logout
- user cannot see private data.

These scenarii should be made by the end user. These are what you will agree with your customer on legitimate behaviour. The customer doesn't care of how much code coverage your unit tests covers. It cares about getting things done.

This is functionnal testing, and should be what your agreed what the software should do, it should also include functionnal domain (speed, acceptable error rates, SLA...). This ensure your software is conform to what your customer is ready to pay. Humans being are acceptable yet expensive bots usable for this tasks.

From this, you can now refine cost assessment, and have the fun part of making it fit in your customers need, and buget.

2.1.6 Architecture

BAD stays good. Your API for interaction is now well defined. It will be a REST API, based on a web framework en kit, or if it is a standalone application on a pseudo REST API on a local virtual database :)

ORM the good, the BAD, the evil

> ORM are tools, the only evil in tools lies in the hand of the craftman.

I have a preference for the following : *ActiveRecord, pymongodb, SqlAlchemy, DBIx::Class*. I have a reluctance to use *doctrine, hibernate, Zend, propaganda*

The 3 aforementioned one are idiomatics in their own langage. Using adapters on the fly, and being very easy to use descriptive.

Normally, your job is one of a very well paid secretary : you type the name, the type of data, and make hybrid properties when some adpatations are needed.

One page per realm, mostly one file for AAA, try to group objects related to one another. If you have avoided cyclic references, you normally have disjoints hierarchies of object.

ORM **abstracts interactions to database** as a results, you do not need to know for which RDBMS you are coding for. Since ORM had great successes with RDBMS, NoSQL ORM like pymongodb have almost the same interface.

Authentication

Here are some of your choices :

- stateful/less authentified connection at socket level ?
- authentication at the applicative layer (login/pass + session) in web application ?
- LDAP, AD, CAS, RADIUS, DIAMETER, Kerberos ?
- System mechanism like PAM ?

Well, a **good** langage is there to abstract the Operating System and the underlying mechanism. A good AAA framework should have one line in a config file to choose the right mechanism.

Some ideas :

Web authentication

- Strong and secure but hard to use is to use TLS client certificate ;
- login password grant a cookie related to a unique ID related to a session with a time to live. Less secure, easier to use ;
- Authenticating proxies (CAS) used in some universities ;
- Web server authentication mechanism (realm based) (supports RADIUS, LDAP ...) ;
- OpenID, facebook connect ...
- PKIs mechanism (ssh (like on github), GPG ...)

GUI authentication

- using a REST API doing a scenario of authentication (refresh the session from time to time) ;
- system authentication on a realm (kerberos, AD, LDAP, PAM) ;
- validating a hash of a password on a stored hash ;
- specific hardware ;

Other authentication

- IP based on a secured, hermetic network ;
- DHCP leasequery based ;
- SIP ;
- HTTP header based (seen, but not my first choice) ;
- Physical standalone terminal, which is in a physically protected area, identification is made by asking one's ID card.

Biometry is **not** acceptable as an authentication mechanism :

- it is measure based, therefore error prone ;
- you leak almost all your biometric data without possibly preventing it : **your biological measure are no secrets, nor are they revokable**. It is not eligible as a secret for an authentication mechanism.

Modern Lightweigth web frameworks are based on layers of decorators.

One layer for TCP / HTTP / cache / routing / session / authentication ....

These layers are interchangeable. This is the boring moment when you have to choose configure in the .ini or YAML if you use memcached, file, memory for your cache mechanism. You choose your DB engine, your authentication mechanism.

In real life, you'll have very important people coming without having any ideas of the model to tell you what to choose. Your boss' son probably thinks is a hacker, and knows if memcached is better than regular server side session, or if mysql is better than postgresql. Let it be, you normaly have taken enough precautions to be prepared to fix most of the design by committee choices.

Having used, framework en kit, leaves you a big lattitude in correcting these king of problem, so you should not care about this. Only remember that auto-increment varies from database to database. So don't make asumption if you insert data at setup on the **logical ID** a record should have.

2.1.7 mVc = now is the time for the views

It is pretty straightforward.

one format to rule them all

JSON has thanks to the savant idiocy of Java and PHP developers becomes the de facto standard.

It is readable in almost all langage, frontend and backend identically.

Your data with a noSQL object will be probably a json. Well, easy.

If you have non cyclic distinct hierarchies of objects, they do fit nicely in tree like structure, so JSON.

Oh, I forgot, almost any good lightweigth web framework has a JSON driver that can change data made of hashables (or dict in python) with just one line (either has a decorator, or as a dynamic view ...).

Error handling

Use error numbers, plus a plain word description, eventually a hashable of arguments usable in a template. The error number must be unique and usable for fetching an internationalised template. The plain text is for the developper, it should be logged if it is critical far from the eyes of the script kiddies. The information needed by a developper (stack trace, auth token), are not the same needed by an end user.

If you have a bug tracking system, the easy way to number errors is to prefix it to a ticket number (bug, evolution...). And if it is a bug number, make sure your unit test has one test (at least) that correlates to this number so that you can check easily if this bug was resolved, or where to find it.

In your JSON always reserve a keyword is not in any possible way the name of an attribute of any business object to store the status.

By convention you can decide that **__status** is reserved for something gone wrong.

Then, you **MUST** decide a consistent policy of error multiplexing and the one that are shown to the customer (critical, error, warnings), the one that are logged.

Logging

Python, Perl, Ruby have excellent abstraction classes for logging. Unices have syslog daemon that may prove interesting because your security alerts should be sent on a machine different than the ones potentially attacked or failing.

If you don't understand this, you are a PHP developper thinking a developper should not care about how the system work. I can lend you a rope.

Error levels

Dont innovate. There are standards.

```
man 3 syslogd
```

HTML fragments

Sometimes it is easier to generate an HTML table server side or a complex HTML than using jquery to construct the view of an object on the fly from a JSON.

HTML fragments * do not have <body></body> * the rendering of rst, wiki, markdown is an HTML fragment ; * they usually do not include stuff like jquery ; * classes are used wisely even if not described by a CSS so that items can be queried ; * *label for=* , *fieldset* are nice HTML tags ; * indent manually (it squashes a lot of bugs) ; * no presentation, this is jquery / CSS job to do the formatting ; * kill developpers puting javascript here : best place to put javascript to be executed when the fragment is loaded, is in the ajax onsuccess callback ;

HTML Pages

By convention you can serve midly templatized web page for the integrators.

Adding a global css, a one refering to the page name. Same goes for the javascript, a nice feature is handling the authentication automagically (and workflow logic based on session / transaction states).

CSV, PDF, Excel, OpenOffice

CSV is a standard, dont invent your own.

Tabular data are easily converted server side in CSV, Excel or PDF. There are a lot of libraries helping you in these tasks. If you want a *nice* output, you will probably have more work than you imagine.

OpenOffice can be generated both server side and client side. It is always best to let the customer's CPU burn instead of your servers.

Static files

It is quite usual in order to have a consistent set of mocking data to include them in the backend repository. There are usually mechanism of caching and reverse proxying to make them served fast by the frontend. It ensures, the delivery of static files are in the loop of the change management.

Functionnal testing : a 3 in 1 operation

Remember the functionnal testing, and transactions scenarii, and mocking data ?

These are your setup data. Since, in the life of your project (that you wish to be successfull) you hope for large batch of data transfer, you need a provisioning mechanism.

These are mechanism involving structured data YAML, JSON, CSV of consistant objects you want to upload.

If you are on the web, you have a REST API. Well, provisioning is as easy as making a user agent that loads JSON in a file sequentially.

And you have scenarii of login/querying/logout that is described.

Code it.

These are the reference implementation of your backend API. They'll be used by the front end to actually make their first interacting web pages.

Dont fire and forget. Integrators might find bugs or inconsitencies. It is not too late to correct your physical model.

Standalone

You need no web server. You need an ORM that will be included.

CHAPTER

THREE

REFERENCES

http://erjjones.github.com/blog/How-I-built-my-blog-in-one-day/

http://toys.lerdorf.com/archives/38-The-no-framework-PHP-MVC-framework.html

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*