Design for Change

Thoughts about Joomla 4 Architecture

Herman Peeren, June 2015, version 0.9

My theme for Joomla 4, from an architectural point of view, is “Design for Change”. Not just make everything flexible enough to change, but even more: make “change” the central theme. Because new, unforseen things will happen.

Event sourcing

Event sourcing (ES) is one of the modern techniques to handle change while keeping simpicity: do not store the changed state, but the event that caused the change. It is a more elaborate model than what I saw from Niels http://nibralab.github.io/joomla-architecture/command-undo.html and Chris (JAB15-presentation http://jandbeyond.org/programme.html?view=session&id=74&return=L3Byb2dyYW1tZS5odG1s) of a command-queue. A command is imperative (describing what has to be done with data), while an event is declarative (describing what the desired outcome is; immutable, because you cannot change the past). These are ideas stemming from the Functional Programming paradigm (not te be confused with procedural programming) and make things simpler and more expressive. Immutability (easier to handle), scaling and modularity are consequences. ES is always combined with CQRS: two different models for reading (retrieving, publishing) and writing (updating, adding).

A well known system that applies the same principles as ES is accounting / bookkeeping: you don't store the new balance of an account, but keep track of all changes that have led to a change in balance. Also: when an invoice has been sent, you don't change it, even if it was a mistake, but you correct it, for instance with a new credit-invoice. When you would only store the current state, you don't have all information available anymore why, how and when the changes were made; with Event Sourcing you have the whole history available and it can also be re-used in other forms.

Some reading:

- Martin Fowler: http://martinfowler.com/eaaDev/EventSourcing.html although that is a bit older and more “command sourcing” than event sourcing
- Vaughn Vernon: “Implementing Domain Driven Design” (I only have it in dead tree and in a DRM-protectect ebook version), Addison-Wesley/Pearson Education 2013
- Scott Millett: “Patterns, Principles and Practices of Domain-Driven Design”, Wrox 2015 (I only have it in dead tree version; only ebook version I saw was Amazon Kindle)
- The DDD/CQRS mailing list (in which Greg Young, who coined Event Sourcing, actively participates): https://groups.google.com/forum/#!forum/dddcqrs
- several videos from Greg Young on ES and CQRS like https://www.youtube.com/watch?v=JHGkaShoyNs , https://www.youtube.com/watch?v=8JKjvY4etTY , https://vimeo.com/31645099
- BTW: January 28 + 29, 2016 DDD-Europe Conference, with a.o. Eric Evans, Vaughn Vernon, Liz Keogh and maybe also Greg Young: http://dddeurope.com/

I see several possibilities vor ES in a CMS / WebApplicationPlatform:

1. most obvious: for **content**. We are talking about versioning and workflow, but you'll more easily get that when you don't store the changed state + log all changes to keep track of it, but model it by means of events, reconstituting to the desired (latest) state in the read model.
2. **cck**: that is a change of a structure. It is new to use ES on such a meta-level. Makes me think of older Lisp-programs that produced programs to produce programs... The modeling of content types is a use case on its own and we can apply ES to that too.
3. **migration** from current situation: design the change, not the changed. This is new too. It is the same idea as we know from TDD: tests are boring to write afterwards; you should do them upfront, so they are used to get your requirements. With migrations it is the same: writing migrations afterwards is boring, so we better define that first (if possible; otherwise: do it per feature/ use-case).

When using ES you normally combine it with CQRS (not necessary the other way around). I saw CQRS on Joomla 4 Requirements spreadsheet https://docs.google.com/a/community.joomla.org/spreadsheets/d/1mAJ2ZdQnv_T1LsKaP5mT0YJ_jJtNgIDaS24WAJoMFpI/edit?usp=sharing somewhere neare the bottom (J4-0028), but it is not somthing you bold onto the system: it is a fundamental architectural chioice whether we go in this direction or not.

A production-ready framework for ES in PHP is Broadway: https://github.com/qandidate-labs/broadway (actively maintained from my hometown). Will provide a POC for Joomla before the meeting in Odense. Also see http://www.slideshare.net/HermanPeeren/event-sourcing-41545888

Making implicit choices explicit: Active Record or Datamapper

In Nicholas' blogposts on http://www.dionysopoulos.me/ I saw the suggestion to use Eloquent, Laravel's ORM. Laravel is using the **Active Record** pattern (see Fowler's book "Patterns of Enterprise Application Architecture"), like Joomla does, and by choosing Eloquent you are implicitly choosing to continue using that pattern. I often had to deal with more complex domains that I had to model on a Joomla platform and Active Record is one of the things that limits that, or at least makes it more complex than nexcessary. Hence my choice for the only full blown **Datamapper** pattern in PHP: Doctrine 2 ORM. Doctrine 1 ORM was also based on Active Record, but development of that was discontinued, because of the limitations for more complex models. Laravel is often seen as a simpler alternative to Symfony. But now more complex system are being built in Laravel you see that even there things like a repository pattern or even using Doctrine 2 ORM are considered. Are we continuing Joomla's ceiling for complexity? Or is Active Record just fine for most of our use-cases?Or can we easily combine both choices? We should at least make our choices explicit.

One of the concepts I like very much in Doctrine 2 ORM is t**ransparent persistency**: the model doesn't know anything about persistence. You build a model, agnostic of any persistence, and the mapping layer takes care of persistence to a database. That can be a SQL-database, but just as well some NoSql. The model doesn't have to change at all when changing the mapping to another kind of database (or other persistence mechanism). No SQL in the model, model and persistence are decoupled!

Content

Content = that what is contained, in a container. Content management in Joomla is both management of content and of containers, two different things, with different requirements. In current Joomla the two are mixed: we use an item from a component (for instance an article) as the main thing on a page and put other things around it. Hence the difference between components and modules in Joomla. Whereas if we see a page as a container that can contain several components, we don't need "modules" as we have them now: those are just component-views (like in Nooku).

I'd like to better split the requirements and use-cases of those two:

- **content management** (+ meta-content management, like categories and tags). Create and edit content.
- **container management**. Pages are containers. Where to show content and how? Nesting of containers.

This has consequences for routing and URLs too: we now use the same URLs for the containers and the content.

In 1965 Christopher Alexander (the architect who coined the term "Design Patterns") wrote a paper "A City is not a Tree". See: http://www.rudi.net/pages/8755 . We often try to model things into mutually exclusive groups, like we do with a tree-structure. A semi-lattice (or directed acyclic graph; a graph like a tree, but with multiple parents) is more accurate for modeling a lot of complex things in our world. A tree would be an oversimplification: "**content is not a tree**".

There is a standard (well, a JSR) for content management in Java: JCR (Java Content Repository) http://www.day.com/specs/jcr/2.0/index.html . It has been ported to PHP, which was initiated by Typo3: PHPCR http://phpcr.github.io/about/ . It was the basis of the Content Management Framework (CMF) in Symfony, building blocks to build a CMS. Some of those CMF-bundles, like routing, are used in Drupal 8. JCR is based on nodes, grouped in a directed acyclic graph (so: not limited to a tree!). If you want to do something with content management, it could be wise to learn something from existing standards.

Behavioural flexibility and DCI

Some people, like James Coplien, say what we do, is not Object Oriented Programming, but Class Oriented Programming: we put all kinds of behaviour (methods) in a class, while only needing some of them in a certain context. Trygve Reensgaug, the same guy who invented MVC back in 1979, started with a new paradigm to do "full object oriented programming": **Data-Context-Interaction** (DCI)

http://fulloo.info/Documents/ ; mailinglist: https://groups.google.com/forum/#!forum/object-composition ; an inspiring video where James Colien (“Cope”) explains it on October 2012 Splash Conference: http://www.infoq.com/presentations/Reflection-OOP-Social . In DCI the behaviour that is needed within the context of a use-case is added to an object only within that context. This is not the same as traits: traits are behaviour added to classes, before they are instantiated. In DCI the behaviour is added at runtime. The objects are said to play a “role” within the context. Just like you play different “roles” (with different behaviour) in different contexts.

One of the basic features of Nooku are mixins. You can use them instead of traits: adding behaviour to objects (runtime) instead of to classes. You can read some critical remarks about traits on this blogpost from Anthony Ferrara from 2011: http://blog.ircmaxell.com/2011/07/are-traits-new-eval.html . Those mixins are fully testable. I'm working on an implementation of DCI using those mixins and want to provide a POC for Joomla before the meeting in Odense. This could be a better approach than the double dispatch (aka visitor pattern) Niels has proposed.

I'm combining those DCI-ideas with Event Sourcing and other ideas from Functional Programming. See http://www.slideshare.net/HermanPeeren/dci-dddbe and http://www.slideshare.net/HermanPeeren/improve-yourphpcodewithfp

Perspectives

In our CMS-system at the moment we only have to deal with just one user at the time. If that one user only gets a model with everything in it that user is allowed to do, then we don't need any ACL while the user is interacting with the system. I call that limited model, just for that particular user, a “perspective”. That model is composed for that particular user in that situation. In that composing-phase you need ACL, but not after the perspective is made; then users can just use anything that is in their models.

REST

Using hypermedia as the engine of application state is also: design for change. You let the clients discover what the webservice has to offer them (like we more-or-less did with WSDLs in SOAP almost 15 years ago...). So that fits into my “design for change” theme.

There is a difference between adding an API to our legacy codebase, as is now planned for Joomla 3.6, and building a RESTful system from the start, like for instance Nooku has done. When building a Nooku component, you get an API with it, without coding anything extra. See http://guides.nooku.org/json.html and this example of using client-side MVC with Backbone.js http://guides.nooku.org/tutorials/backbonejs-nooku-together.html . Our strategy will probably be to first, in Joomla 3.6 build an API on top of the existing Joomla codebase and then to refactor the underlying engine for Joomla 4 without changing the API. Using client-side MVC brings us also closer to a Service Oriented Architecture (SOA).

Core of REST is the Uniform Interface: not different methods (remote procedures) you can call, but a limited set of methods (most often CRUD) on different resources, referenced by a URL. The other side of our system, the persistence in a database, also essentially uses CRUD to store the data. BUT: that doesn't necessarily mean that the middle part, the model, also has to be limited to just CRUD. That would yield a so called “anemic” model, whereas in DDD we try to model more “rich” behaviour. That means that we not only on the side of the database have to have some mapping to our model (an ORM), but also on the side of the Webservices. Also see http://www.martinfowler.com/bliki/AnemicDomainModel.html and http://verraes.net/2013/04/decoupling-symfony2-forms-from-entities/

A very good book about REST, I only have in dead tree version is “RESTful Web Services” by Leonard Richardson (yes, the guy who invented the Richardson Maturity Model) and Sam Ruby, O'Reilly 2007. A newer book, “RESTful Web APIs” by Leonard Richardson and Mike Amundsen, was published by O'Reilly in 2013. The later book I have in PDF, so I could share.

BTW, for who has no subscription: the June edition of php[architect] magazine is free and is about APIs: http://www.phparch.com/magazine/2015-2/june/

A still vague thought I have is a SuperAPI: WP has an API http://wp-api.org/ and so have Drupal 8 https://drupalize.me/blog/201401/introduction-restful-web-services-drupal-8 and Magento http://www.magentocommerce.com/api/rest/introduction.html and probably more related CMS-like web

applications. Mabe we can do something with adapters to make a super-API with which you can manage different sites from a Joomla-site. This is more about consuming other APIs than about building our own, but still got the feeling that there are possibilities to add something interesting to the market.

Components, modularity and HMVC

In Nooku Platform everything is a component (including the application). No need for modules (those are just views) or plugins (those are just methods of components that are subscribed to an event). I like reusable components (components in other components, like com_categories in Joomla or com_files in Nooku). It gives modularity, reuse of code, and avoids duplicate code.

A problem with HMVC is that some structures don't belong in views, but in models. For instance, if an order consists of orderlines, then those orderlines don't belong in some HMVC-structure, but are part of the order-model. This is something that is also related to both an ORM and the container-content difference.

Hooks, aspect oriented, and events

I saw a proposal to put onBefore... and onAfter... around several methods (like we see in FOF and F0F). The problem with it is that that is always limited to the methods where you put that upfront. In Nooku and in Typo3 we see more flexible solutions where you can add such “hooks” around ***any*** method.

One application, no separate administrator

Like I mentioned in my presentation on JAB11 http://www.slideshare.net/HermanPeeren/joomla20-architecture since we have ACL we can make the administrator part of the whole application. As mentioned in some of the discussions recently we need to namespace the CMS for that, because now the administrator component classes have the same name as the ones in the frontend. The /administrator can be a route of the main application. It is probably possible to gradually work that into the core CMS, while maintaining backwards compatibility, but it would a challenge to make that also possible while other extensions keep working too.

BDD

Behavioural Driven Development (BDD) is an extension of Test Driven Development (TDD): it works with functional tests and has more connection with use-cases and features. It can be done together with non-technical people, always giving “customer value”. It can avoid developers diving into some abstract technical exercise and loosing contact with the real purposes and requirements. It is exactly therefore that I'm even more a fan of Behat than of Codeception. I'd like to come to a good workflow, that fits together: use-cases/scenarios → Behat tests → red → implement a DCI Context→ green → refactor. Also see my JAB14 talk http://www.slideshare.net/HermanPeeren/next-generation-joomla

Routing

Routing can be more generally solved by means of a context sensitive grammar (which can be split into two context free grammars). The current Joomla routing would be a special case of such a general solution and can be used for backward compatibility.

DBAL

Joomla has its own Database Abstraction Layer (DBAL) but that has many problems, especially with respect to maintainability. How much resources do we spend to keeping MS SQL Server or PostgreSql in our package? This can be easily solved by not keep on turning our own wheel for that. For instance Doctrine DBAL is an solution that is long time stable, much used and much wider in possibilities. Another good thing with it, especially when used in combination with Doctrine ORM, is that you can create database schemas and content on one spot and have it automatically generated for all supported databases. And a package like Doctrine Migrations can also be used to maintain upgrades and changes in the database schemas http://docs.doctrine-project.org/projects/doctrine-migrations/en/latest/reference/introduction.html

Some remarks bout the GoF Design Patterns (DP)

Shortly after the “Gang of Four” (GoF) book about design pattterns was published in 1994 some people showed most patterns were not needed in a programming language like Lisp. Design patterns are common solutions to common problems and the problems in that book are related to the object oriented paradigm (or maybe I should say: class oriented). The fuctional programming paradigm does not have the same problems.

Marco mentioned some books about those classical design patterns. If you don't want to read a lot of books, but prefer a short introduction, you could also read the book of Brandon Savage: “Practical PHP Design Patterns”, http://www.brandonsavage.net/design-patterns-dont-have-to-suck/ (have it in PDF).

If you don't like design paterns ;-) or maybe to understand them better, you could have a look how Anthony Ferrara uses Occam's Razor to eliminate the need for most design patterns in PHP in this blog post: http://blog.ircmaxell.com/2013/09/beyond-design-patterns.html