Journal of Physics: Conference Series **664** (2015) 052012 doi:10.1088/1742-6596/664/5/052012

Indico – the Road to 2.0

P Ferreira, A Avilés, J Dafflon, A Mönnich, I Trichopoulos

513–1–008, CERN CH-1211, Genève 23, Switzerland

E-mail: indico-team@cern.ch

Abstract. Indico has come a long way since it was first used to organize CHEP 2004. More than ten years of development have brought new features and projects, widening the application's feature set and enabling event organizers to work even more efficiently. While that has boosted the tool's usage and facilitated its adoption by a remarkable 300,000 events (at CERN only), it has also generated a whole new range of challenges, which have been the target of the team's attention for the last 2 years. One of them was that of scalability and the maintainability of the current database solution (ZODB). After careful consideration, the decision was taken to move away from ZODB to PostgreSQL, a relational and widely-adopted solution that will permit the development of a more ambitious feature set as well as improved performance and scalability. A change of this type is by no means trivial in nature and requires the refactoring of most backend code as well as the full rewrite of significant portions of it. We are taking this opportunity to modernize Indico, by employing standard web modules, technologies and concepts that not only make development and maintenance easier but also constitute an upgrade to Indico's stack. The first results are already visible since August 2014, with the full migration of the Room Booking module to the new paradigm. In this paper we explain what has been done so far in the context of this ambitious migration, what have been the main findings and challenges, as well as the main technologies and concepts that will constitute the foundation of the resultant Indico 2.0.

1. Introduction

Indico[1] was born of a European Project, a joint initiative of CERN, SISSA, University of Udine, TNO, and Univ. of Amsterdam. The main objective was to create a web-based, multi-platform event storage and management system. This software product would allow the storage of documents and metadata related to scientific conferences and workshops. The project started in May 2002, and ended 2 years later. After the end of the European Project, CERN took over the core modules that had been developed internally and put them together in a software platform aimed at fulfilling its own needs. Indico started as an Open Source project under the GPL version 2 (and later version 3) and so it has remained to this day – the entirety of its code is available online, through the project's GitHub repository[2].

Over the years, Indico has evolved into a more general event management solution. Although its initial goal was to provide conference organizers with a set of tools that could help them through the entire conference life cycle, this initial feature set was extended to other organizational events (such as meetings and lectures). It has since grown to include other functionalities such as a full-fledged Room Booking module. In 2009 Indico became CERN's

[1] Originally spelled *InDiCo*, or "Integrated Digital Conference"

[2] http://github.com/indico/indico

official hub for collaborative tools, providing a common user interface for video-conferencing, chat and webcasting/recording systems. As the project grew, other organizations in the High Energy Physics realm started using it, effectively creating a network of more than 150 different servers, distributed across 4 continents. The Indico Community keeps growing day by day. At the time of the writing of this article, Indico's latest release is version 1.2.

More information about the features that Indico provides can be obtained through its official website[3].

Past debts

It's been more than two decades since Ward Cunningham first coined the term "Technical debt"[1]. The power of this well-known software development metaphor may not be obvious at first, and its main message is for sure not unknown to virtually any software developer: bad code generates more bad code and the sooner it is stopped the better. But its uniqueness comes from the scenarios that can be built from it: software developers borrow time by writing lower quality code and/or delaying the "payment" of past debts (e.g. by postponing a refactor/rewrite). This "borrowed time" will have to be eventually paid back with higher quality code. Software projects that pay upfront will not benefit from the advantage of an initial "loan", while the ones that postpone their payments will instead have to contract loan after loan, many times risking "bankrupcy" (non-maintainability). Few metaphors can be as powerful as this one.

The bulk of Indico's source code was developed between the years 2002 and 2008. This is what is now called *MaKaC*[4] or simply "legacy code". This layer was expanded upon in 2008, when a series of AJAX-powered interfaces were added to the application, opening way to version 0.97[2]. By this time, a plugin system was added on top, which allowed for new code to be written without having to be placed within the application core – this has positively contributed to the maintainability aspect. However, between 2008 and 2011 Indico experienced an explosive growth in the amount of functionality provided. It was also by this time that the development team consolidated itself, with longer term resources assigned to the project and standard development practices as well as software testing finally in place. The price of this growth was, however, partially paid with technical debt. During this period, Indico greatly benefited from the contributions of students from different schools. Those were mostly junior developers who would work on a single sub-project or task. This "debt" was essential in effectively integrating those contributions into Indico's source base. The peak of the number of Python lines of code was attained in 2012 (160 kSLOC) and it has been decreasing ever since (115 kSLOC at the time of writing).

Code quality-related debt is common in most software projects – the fact that it is something really hard to measure means that it is hard to estimate its prevalence, but very few software projects can say that their code base is free of imperfections or "quick and dirty" fixes. Indico, of course, is no exception. But it differentiates itself from most software projects in a fundamental issue which underlies a great share of the recent "loans".

Indico was from the beginning experimental in nature. That meant, among other things, using technologies that were less conventional back in those days. Such an example is the choice of Python as the main programming language for the project. This seems to have been a good choice, since Python has since become one of the most popular programming languages in the world[3] and frameworks such as Django[5] and Flask[6] became essential tools in the lives of many

[3] `http://indico-software.org`

[4] `MaKaC` is the original name of the module that CERN contributed to the European Project and eventually became to mean "Indico"

[5] `http://www.djangoproject.com`

[6] `http://flask.pocoo.org`

web developers. A more controversial choice was that of the ZODB[7] as main storage for the application. The golden age of object-oriented databases (the late 90s and early 2000s) seems to have passed, and the excitement around those products has naturally cooled down. Despite being a remarkable piece of software engineering (supporting ACID transactions, history and blob storage), the ZODB seems to be, at the time of writing, a niche product with a small and potentially shrinking user base. Its 100% language-dependant approach may have contributed to this, as well as its lack of built-in indices and ad-hoc queries.

1.1. The database problem

It is impossible to reject the role of the ZODB in the technical debt that Indico accrued over the years – after all, it works transparently with seemingly "native" Python objects, both its main advantage and its greatest flaw. Choosing the ZODB simply meant rejecting any possible isolation between database and business logic at the code layer. This closed the door for an easy exit strategy and limited the directions the development could be steered into. At the same time, Indico has shown to evolve in directions that steered away from what can be considered ZODB's main playing field: large-scale storage of separate objects that require a limited number of query operations.

The ZODB posed other problems as well, such as the lack, for many years, of an affordable solution for replication. ZRS ("ZODB Replicated Storage"") was not open-sourced until mid-2013, when it was already clear that an alternative had to be found. In addition to that, the small size of the ZODB community meant that very few 3rd party tools were available and the documentation was pretty scarce, which made the job of new and potential contributors much harder than with competing technologies.

2. The plan

In 2014, Indico finally got the resources that it needed in order to start paying its long-standing technical debt. The aim was to finally abandon the ZODB and slowly move to a PostgreSQL-based solution. SQLAlchemy[8], a database-agnostic solution for Object-Relational Mapping (ORM) was chosen as the insulation layer between business logic and database. Throughout this process, several things would be changed – a move of this dimension would mean rewriting tens of thousands of lines of code and possibly whole application modules. It would have been a waste of resources and a lost opportunity if this were limited to a simple database "port" without any kind of modifications in the underlying technology – this was the moment to change things, and that was taken very seriously into account.

The plan would consist of a cycle of intermediate releases that would happen between the last release of the fully ZODB-based Indico (version 1.2) and the new PostgreSQL-based system. Those would be minor releases of the `1.9` branch, short-lived, in intervals that would range (with the exception of the first two) between 1 and 2.5 months. Indico's development practices could comfortably accommodate this kind of process – a highly agile environment combined with permanent code reviews and testing as well as iterative steps based on small tasks and constant re-assessment of progress and priorities.

During this time, CERN's Indico instance would be using a dual-DB setup, with PostgreSQL taking a larger and larger share of data with each successive release. The complexity of this setup was known to be significant from the beginning – commits would be happening on two completely different databases at the same time and data from heterogeneous backends/technologies would have to be joined together. This is the main reason why it was decided that throughout the "migration period" there would be no public releases of Indico. In the end, Indico adopters

[7] `http://www.zodb.org`
[8] `http://www.sqlalchemy.org`

would just see a version `2.0` that would "magically" know how to migrate their data to the new backend.

3. Challenges and Solutions

Running a dual-database setup in a production environment as demanding as CERN's is by no means an easy task, even more so if this implies constant changes in both databases, with each incremental release offloading an increasingly greater share of data on to PostgreSQL. There are several issues that, in Indico's case, constitute important challenges:

(i) Commit synchronization – database commits need to be simultaneous, and rolled back if a problem happens in either store;

(ii) Data migration – Information contained within ZODB objects needs to be migrated to a relational structure in an automated way. Other Indico instances should be able to use this mechanism without major issues;

(iii) Diversity of schemas – Different versions of Indico have different "schemas"[9]. The migration mechanism should be able to migrate any Indico instance to the new backend;

(iv) Diversity of relational schemas – There would also exist different versions of the relational schema (one per intermediate release and then for each future version) which would need to be upgraded in an easy and transparent manner;

(v) Cross-database references – ZODB objects are highly interconnected. At all times it is necessary to have PostgresSQL tables referencing ZODB objects and vice-versa.

Fortunately, there was already a solution for *i*: a small third-party library called `zope.sqlalchemy`[10] provides synchronization between the "Zope Transaction Manager" (which manages commit and roll-back operations in ZODB) and SQLAlchemy, thus allowing dual-DB operations to happen in a transparent manner.

There was, however, a need for better tools that could turn *ii* and *iii* into non-issues. Since version `0.98`, a "migration script" is distributed with Indico, which is capable of upgrading the "schema" of a sufficiently recent yet older version of the Indico database to the latest one. This started as an attempt to unify all previously existing partial migration scripts that had to be run in a specific order so as to provide the user with the desired outcome. It would automatically detect which migration steps were needed and execute them after user confirmation. This mechanism was not scalable for an operation that required cross-database data migration, not to mention that the "highly embedded nature" of ZODB would make it very difficult to use it after the actual ZODB object definitions were removed from the code base. A simple yet effective migration mechanism was then developed, called `zodbimport`, which is capable of inspecting the ZODB database root even in the presence of broken persistent objects, thus enabling the developer to write highly modular "translation code" or "importers" that create the corresponding data structures in the relational backend. This mechanism is extensible to Indico plugins and announces itself through `setuptools`' entry point mechanism. It is fully integrated with Indico's existing Command-Line Interface (CLI).

Solving *iv* was once again facilitated by the existence of a third party Python package, `alembic`[11], that integrates with SQLAlchemy as to provide bi-directional migration between different versions of a database schema. The migration scripts can be auto-generated to a very helpful degree of accuracy. The migration process was also integrated into Indico's CLI: `indico`

[9] Since the ZODB is an OODBMS, by "schema" we mean a loose definition of the hierarchical structure of ZODB persistent objects.

[10] `https://github.com/zopefoundation/zope.sqlalchemy/`

[11] `https://alembic.readthedocs.org/`

`db upgrade`, for instance, will attempt to upgrade the existing database schema to the latest version available.

Finally, the topic of cross-database references (v), one of the most complex in terms of possible ramifications of its behaviour, ended up being solved through the combination of weak references, proxy objects and adapters.

SQLAlchemy → ZODB references were easily tackled through the use of weak references. For instance, an entry of the `rooms` table (or corresponding SQLAlchemy object `Room`) would contain a column `owner_id` that would hold the identifier for an existing Indico user. The downside of this approach is evident: cascading cannot be implicitly applied and there is the danger for "dangling references". Careful inspection of the code and intensive testing assured that no "loose ends" were left behind – in reality, Indico users are never removed (only potentially merged), but the problem could have been easily solved through explicit re-allocation of the room to another user, its "orphaning", or simply its deletion.

ZODB → SQLAlchemy references were obviously harder to work around, since ZODB data is by nature object-oriented and schema changes come at a very high cost. Simply replacing the reference to a "user object" with the corresponding ID would take several hours, only for the ensemble of events contained within CERN's Indico database. Typically, such an operation would instead take more than a half-day, since user objects are referenced from several other objects in addition to events/conferences: things such as contributions, materials and access control lists (in growing order of nesting). Going over the whole ZODB and basically rewriting it would be just infeasible. However, it is known (and logical) that ZODB saves the state of each object that it has to store. This means the actual object attributes, but none of the code contained in its methods. In addition to that, it provides an option that can override the class of any object, right after loading it, with another one. This said, an `Avatar` object (which represents a user) could be easily replaced with an `AvatarUserWrapper` at load time, thus allowing for completely different behaviour. In this case the `AvatarUserWrapper` provides the same interface of an `Avatar` object but ignores all ZODB-stored information (except for the user ID), fetching it instead from SQLAlchemy/PostgreSQL. This ingenious application of the "Adapter" design pattern[4] allows legacy code to work with users as if they were still stored within ZODB while they are actually already stored in the new backend. Legacy code will be eventually replaced, as the migration effort progresses, thus rendering those objects unnecessary.

Non-technical aspects

As easily noticeable, the technical part of this operation is already complex enough. But there are other factors that impose significant restrictions on the work that is being done and increase the amount of uncertainty involved therein.

The first point is the inherent difficulty in producing time estimates for such a project: even though the objectives are clear, the percentage of the project's code base involved is very large and the nature of the work being done (refactoring, sometimes rewriting) adds an extra dimension of indefiniteness. While traditional agile development tries to split the problem domain in user stories that can then be estimated in greater detail[5], this kind of work consists of sometimes "monolithic" tasks that cannot even be performed in parallel. Another important factor is that we sometimes are working in "unchartered territory", with technologies and libraries that were previously unknown to us in detail. In a team that has to be agile and responsive to change, it is imperative, in this kind of context, to make sure that information circulates and everyone is aware not only of the new technologies that are being used but as well of the code and data structures that were developed. This is why we have been reinforcing the practice of pair-programming, at least at the beginning of each iteration/sprint. After initial work in pairs, the team then splits the remaining work in smaller tasks and starts working on them in parallel. This has proven to be effective as a means of knowledge transfer.

The second point is the contradiction between the need for this project to last as little as possible (for several reasons, among which the lack of external releases and the existence of limited resources) and the nature of the work being done, highly iterative in nature and requiring frequent deployment. An interval between releases of one month can be quite violent when it comes to the amount of work involved, since there is the need for testing and consolidation, which itself could take that amount of time in a normal development cycle. There is, thus, the risk of delays being introduced – working with such limited time substantially decreases any error margin that may exist.

Finally, it is hard to manage user expectations with such a tight release schedule. Not only the fact that there is a lack of development time for tasks that would normally be targeted is at play: the difficulty of accomplishing such tasks highly increases when they happen in parallel to such important changes. It is, then, very important to synchronise such changes, when they are feasible, with the ongoing changes. This is something that we've tried to do and has already provided us with some "easy wins".

4. Progress

`1.9.0` – A first step

On the 13th of August 2014, a new version of Indico was deployed in production at CERN. It included a fully PostgreSQL-based Room Booking module and a new software stack based on Flask, SQLAlchemy, Jinja[12], WTForms[13] and other smaller 3rd party libraries that are now standards in the Python web development scene. This release (`1.9.0`) was the first known attempt at running ZODB and PostgreSQL in parallel in a production environment. The result was extremely positive, with no serious issues reported and an immediate positive impact in the performance of the room booking module.

`1.9.1` – An extensible Indico

On the 3rd of March 2015, version `1.9.1` reached production. It was once again a large development effort, which saw a new, more extensible, plugin system be added on top of Indico. This new system is a significant evolution as compared with the old one. In addition to being completely based on the blinker[14] signalling library, it allows plugins to define their own code modules, URLs, templates, static resources, translation libraries and even documentation. This was only partially possible before.

As fig. 1 goes on to show, the evolution that the code base suffered from `1.2` to `1.9.1` alone has been tremendous: not only the amount of legacy code decreased substantially (in about 36%) – the total amount of code itself also shrunk quite considerably (around 15%).

`1.9.2` – The first large-scale migration

Version `1.9.2` marks the first step into Indico's core, this time targeting user information. All user accounts will be migrated from ZODB to PostgreSQL. At the time of writing, the release is nearing completion and will be installed in production at CERN on the 4th of May. Some of the technical issues that we bumped into as well as solutions for them have been already documented above. This is also the first release of a series of six (up to `1.9.7`) that will be very short in duration.

An overview of migrated modules and those that still have to be addressed may be seen in fig. 2.

[12] `http://jinja.pocoo.org`
[13] `http://wtforms.simplecodes.com`
[14] `https://pythonhosted.org/blinker/`

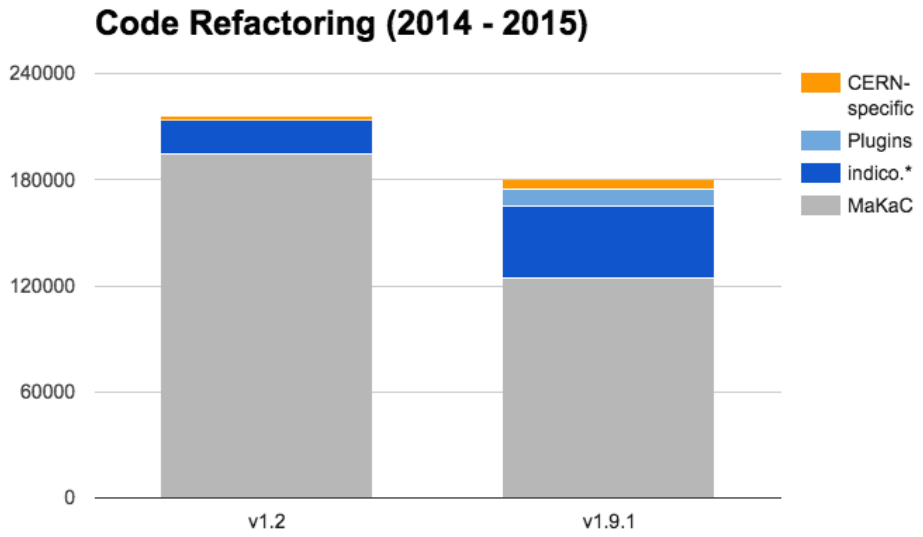

Figure 1. Evolution of the distribution of lines of (Python) code in Indico

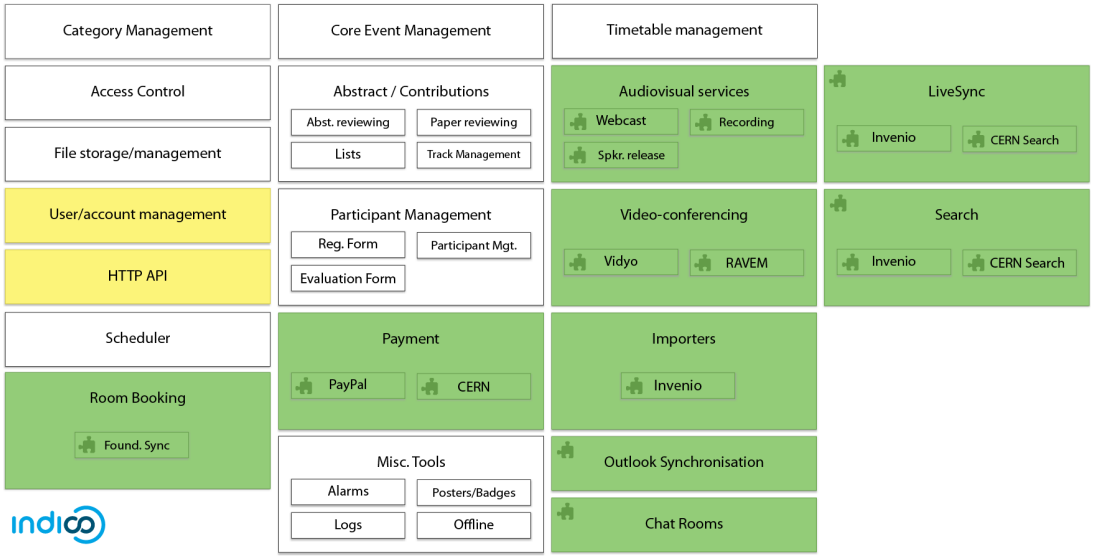

Figure 2. Modules that are already PostgreSQL-enabled (green) vs. legacy code (white). Modules currently targeted are highlighted in yellow.

Plans

A series of additional releases have been already scheduled for the rest of the year. Those are the stepping stones that will eventually lead to a fully-polished Indico 2.0:

- 1.9.3 (May) – Task queue, HTTP API and statistics fully migrated to PostgreSQL;
- 1.9.4 (July) – All references to material files in the new DB, as well as event logs;
- 1.9.5 (August) – Participant and registrant management fully stored in PostgreSQL;
- 1.9.6 (September) – Abstract and paper reviewing, timetable structure (contributions/sessions);
- 1.9.7 (December) – Category and event storage, protection scheme.

The public release of Indico 2.0 is foreseen for the first quarter of 2015.

5. Conclusion

Indico has come a long way since its early days more than 10 years ago. It has grown to be a full-featured application, an indispensable tool in the lives of many event organisers. This growth brought with it new technical challenges as well as the need to move forward and rework some of the pillars of the application. Change has always been present in the project and new technologies have been welcomed one by one, but this database migration is unlike anything before – its short release cycles are filled with intensive periods of work on specific areas of the application.

We are conscious of how daring this operation is and the risks that it entails. Rather than avoiding the latter completely, we have decided to control them – contingency plans are a mandatory part of the carefully-planned delivery process that is in motion. More than that, this process is being constantly reassessed, as we gradually "peel off" layer after layer.

So far, the results have been remarkably positive. The two already complete migrations were long but happened with no problems. No technical issues worth being mentioned in this document have been noticed. There certainly have been challenges, most of them related to the large scale of the code base (which reached almost 160.000 lines of Python code alone, in 2012) and the technical complexity of the operation (mostly due to its dual-database nature and the large amount of data structures). However, the Team has managed to overcome all of those and to deliver version after version. More than that, the code base has shrunk considerably in size, as older code is replaced with better quality and simpler modules. Legacy structures are being quickly and effectively phased out, opening the way to better and more manageable implementations.

Indico seems to be sailing smoothly towards version `2.0`. Waters will certainly be "wavier" ahead, as we go deeper into the application core, but we are confident that our experience and powerful tools will allow us to finish this journey in due time.

References

[1] Cunningham W 1992 The WyCash Portfolio Management System *SIGPLAN OOPS Mess., April 1993* ACM, New York, NY, USA

[2] Gonzalez Lopez, J B and Ferreira, J P and Baron, T 2010 Indico Central - Events Organisation, Ergonomics and Collaboration Tools Integration *J. Phys.: Conf. Ser. 219 (2010) 082002* `doi:10.1088/1742-6596/219/8/082002`

[3] StackOverflow 2015 Developer Survey, available at `http://stackoverflow.com/research/developer-survey-2015`

[4] Freeman E and Freeman E and Bates B and Sierra, K 2004 Head First Design Patterns *O'Reilly*

[5] Shore J and Warden S 2007 The Art of Agile Development *O'Reilly*