PAPER • OPEN ACCESS

Indico 2.0 – the whole Iceberg

To cite this article: A Mönnich *et al* 2017 *J. Phys.: Conf. Ser.* **898** 102017

View the article online for updates and enhancements.

Related content

- Indico — the Road to 2.0
  P Ferreira, A Avilés, J Dafflon et al.
- Indico 1.0
  J B Gonzalez Lopez, A Avilés, T Baron et al.
- Indico: A Collaboration Hub
  P Ferreira, T Baron, C Bossy et al.

IOP ebooks™

Bringing you innovative digital publishing with leading voices to create your essential collection of books in STEM research.

Start exploring the collection - download the first chapter of every title for free.

This content was downloaded from IP address 131.169.5.251 on 29/04/2019 at 23:16

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 102017 doi:10.1088/1742-6596/898/10/102017

Indico 2.0 – the whole Iceberg

A Mönnich, A Avilés, P Ferreira, M Kolodziejski, I Trichopoulos and F Vessaz

28–1–009, CERN, CH-1211, Geneva 23, Switzerland

E-mail: `indico-team@cern.ch`

Abstract. The last two years have been atypical to the Indico community, as the development team undertook an extensive rewrite of the application and deployed no less than 9 major releases of the system. Users at CERN have had the opportunity to experience the results of this ambitious endeavour. They have only seen, however, the "tip of the iceberg".

Indico 2.0 employs a completely new stack, leveraging open source packages in order to provide a web application that is not only more feature-rich but, more importantly, builds on a solid foundation of modern technologies and patterns. But this milestone represents not only a complete change in technology – it is also an important step in terms of user experience and usability that opens the way to many potential improvements in the years to come.

In this article, we will describe the technology and all the different dimensions in which Indico 2.0 constitutes an evolution vis-à-vis its predecessor and what it can provide to users and server administrators alike. We will go over all major system features and explain what has changed, the reasoning behind the most significant modifications and the new possibilities that they pave the way for.

1. Introduction

Indico[1] was born of a European Project, a joint initiative of CERN, SISSA, University of Udine, TNO, and University 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 European Project started in May 2002, and ended two years later. Afterwards, 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 evolved into a more general event management solution. The initial goal was providing conference organizers with a set of tools that helped them throughout the entire conference life cycle. This initial feature set was extended to cover other organizational events, such as meetings and lectures, as well as a full-fledged room booking system. Since 2009 Indico has become CERN's official hub for collaborative tools, and provides a common user interface for video-conferencing and webcasting/recording systems. Meanwhile, the Indico community has been growing day by day, as other organizations in the High Energy Physics

[1] Originally spelled *InDiCo*, or "Integrated Digital Conference"
[2] `https://github.com/indico/indico`

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

realm started using the software. At the date of this writing, there is a network of more than 180 different Indico instances, distributed all over the world. Currently, Indico's latest "stable" release is version 1.2.1 and the bleeding-edge version used at CERN is at version 1.9.9.

More information about the features Indico provides can be found on its official website[3].

1.1. The Legacy

The development of Indico started 15 years ago and most of its core was developed during the first few years. This part of Indico is known as "Legacy Indico" among the current Indico developers. Details on its evolution and also a more in-depth explanation on the technical debt that comes along with the legacy codebase can be found in [1]. Over time, the Python language became more powerful[4] and better libraries[5] were developed. Since they didn't exist back then, the obvious solution was to write them in-house.

Mixing old and new code is not always easy and it often forces developers to make trade-offs, i.e. writing lower-quality code to interoperate with the legacy framework. Another issue was the high fluctuation in the development team as many developers were students with little previous development experience who only stayed for about one year. These two things combined with the lack of established development processes resulted in lower-quality code ending up in the codebase and remaining there for years.

1.2. ZODB

At the beginning, Indico was conceived as an experimental project, and ZODB[6] was very practical for quick prototyping. ZODB, being a Python object database, allows storing Python objects directly, without having to define a structure or use SQL. For Indico, having some instances with hundreds of thousands of objects of the same type, the lack of indexes in ZODB was a major issue. Searching for an object based on some attribute requires iterating over all objects and checking them one by one, which results in unacceptable performance for big collections. The common workaround in ZODB is to maintain custom indexes dictionary-like objects that map some attribute to an object or a list of objects, making it the application's responsibility to keep these indexes synchronized. Also, creating a new index always requires iterating over all the objects to populate it, which may take a long time.

Finally, another problem with ZODB is the fact that it is a niche product, so it lacks a large community and the third-party tools and support that usually comes with it.

1.3. User Interface

Indico's user interface (UI) was initially completely static, without any JavaScript nor AJAX[7]-based interactions. While this was the standard back then, nobody would consider this a good user interface for a web application nowadays. After some years, AJAX functionality was added to various parts of Indico and the UI was improved, but it did not keep up with newer web technologies and patterns, resulting in a somewhat inconsistent and outdated look and feel. Although small UI improvements were made, there was never a good opportunity to completely revamp large parts of the UI as most development effort went into adding new features.

[3] `http://indico-software.org`
[4] `https://www.python.org/dev/peps/pep-0000/`
[5] `https://pypi.python.org/pypi`
[6] `http://www.zodb.org`
[7] Asynchronous JavaScript and XML

2. The Solution

In 2014 the Indico project finally received the resources required to abandon ZODB and move to PostgreSQL, leveraging the SQLAlchemy[8] object-relational mapper to keep using Python objects instead of having to write SQL queries.

The initial idea was to focus on only replacing the database backend, without having to change everything related to it, i.e. controllers, templates and all the frontend code. Soon after starting the migration of the first module, it became apparent that this strategy would take more development effort than re-implementing at least the controllers from scratch. This looked promising as a side-effect of rewriting the controllers would be ending up with a much more maintainable codebase.

This solution, however, was still not perfect. Keeping old templates meant staying with the Mako template engine which lacks advanced features and also has an inconsistent syntax[9]. Mako also makes it easy to blur the boundary between display and business logic by allowing raw Python code inside templates which is something that should be avoided.

As of version 1.9.1, the migration finally turned into what could be called a rewrite. Besides writing SQLAlchemy models to store data and rewriting controllers, all involved templates were rewritten using the Jinja2 template engine. Since none of the old HTML code was re-used in this process, this provided a perfect opportunity to write new JavaScript code and CSS stylesheets using the SASS preprocessor.

2.1. Ensuring Extensibility

Indico is Open Source, so anyone is allowed to modify its code to adapt the system to their own needs. However, modifying the core of Indico does come with some caveats that show up when trying to install official updates.

Updating an unmodified Indico 2.0 instance will be as easy as running `pip install -U indico` and `indico db upgrade` in the terminal. To update an Indico instance with code customizations will require building a new Python package containing both upstream and local changes before installing it. It is worth mentioning there may be conflicts between local and upstream changes that would need to be resolved manually.

To avoid this problem Indico 2.0 will provide a plugin system and hooks in various places throughout the core. That way, custom extensions to Indico's functionality can reside fully within a separate plugin package that will be unaffected by changes in Indico's core. It is obviously impossible to cover all possible places where someone might want to run plugin code, but new plugin hooks can be easily implemented and released quickly in minor releases.

2.2. Complex – Not Complicated

In the current development version v1.9.10, the database consists of about 125 tables categorized into 11 schemas and there are around 275 foreign key relationships between these tables. This is a high number compared to other web applications. As a comparison, the Stack Overflow Q&A website has around 30 tables[10] and GitLab has 80 tables and only 12 foreign keys.

The high complexity of the Indico database is a consequence of Indico providing many features that are distinct enough to justify having separate tables. This actually prevents the database structure from getting complicated as there is just one table backing each business object. Foreign keys do not make the database more complicated either, as they just enforce consistency, something that was impossible to do previously with ZODB.

[8] `http://www.sqlalchemy.org/`
[9] `http://docs.makotemplates.org/en/latest/syntax.html`
[10] Based on a public database dump, which may or may not reflect the live database – https://data.stackexchange.com/

2.3. Benefits

Besides consistency and using a popular, well-supported database there have also been other benefits migrating Indico to PostgreSQL.

2.3.1. Deletion Users sometimes delete items accidentally. It is important for this reason that a deleted object can be recovered. With ZODB, deletions are handled in one of two ways:

Hard deletion: Removing an object from the database when deleted by a user. This is similar to deleting a file using a command like `rm`. Once deleted, it is gone and cannot be recovered except by restoring a backup of the whole Indico system, losing all the data that was changed in the meantime.

Soft deletion: This alternative strategy involves removing the object from anything referencing it and, at the same time, adding it to a collection of deleted objects, called `TrashcanManager` in Indico. Recovery of deleted objects is possible by iterating over the whole collection until finding the desired ones. This strategy is not only extremely slow but also tricky or even nearly impossible depending on the object. This shortcoming is not ZODB's fault. When the trashcan system was added to Indico, the database was still small and the number of items in the trashcan low. After a decade of objects being deleted and Indico usage increasing, the trashcan has grown to an unmaintainable size.

With the new database schema, on the other hand, soft-deletion can be achieved in a much more straightforward way by simply indicating that a row should be considered deleted. Code-wise, there is no need for many changes when using SQLAlchemy relationships as they can be configured to use custom JOIN criteria which automatically filter out deleted objects. Unfortunately, this is not always so straightforward. For instance, when having nested structures, anything that is a child of a deleted element should also be considered deleted. Cascading the deletion flag to all child elements is not an option since this would make exact recovery impossible: Some children may have been deleted beforehand and undoing deletion should not bring them back. Instead, to determine whether a particular object should be considered deleted, we recursively check if it is descending from something that is explicitly marked as deleted. This does not add any significant overhead, since when displaying information stored in an object, its parents are usually retrieved as well either for display or access check purposes.

2.3.2. Damage Recovery Users may also accidentally overwrite data, e.g. by editing the wrong event. Modifications cannot be undone as easily as soft-deletions so, if an event got damaged that way, it may be necessary to recover the data from a backup manually. With a cleanly structured SQL database this is an easy process since there is no need to dig through deeply nested Python structures as it is the case with ZODB. Instead, the backup can be quickly restored to a separate database and data manually retrieved from it.

2.3.3. Speed Like any RDBMS[11], PostgreSQL supports indexes. These allow fast filtering of data using any column that is covered by an index, like, for instance, finding a user by their email address. Having many indexes does not negatively affect performance as it only requires more space on the database server. This is not an issue with both disk space and RAM being cheap nowadays and the Indico database at CERN having a footprint of about 10 GB. Unlike ZODB indexes, which are implemented at the application level, relational database indexes are always up to date and cannot become inconsistent due to application bugs. Such inconsistencies have been common in legacy Indico.

[11] Relational database management system

PostgreSQL also provides advanced index types which can speed up more complex comparisons such as fuzzy, substring, or full-text searches. Indico makes use of such indexes, for example, when searching for users and a search for *monnich* would also find *Mönnich* even though it not only differs in casing but also in diacritics.

Having a fast database also makes reporting and statistics retrieval much easier, often even fast enough to calculate statistics on the fly instead of having to cache them. An example: getting the number of file attachments that are neither deleted nor belong to a deleted object from all events within CERN's production Indico instance[12] takes only about five seconds. Even when filtering by events belonging to a specific large category (which makes the query more expensive), it still takes only around 15 seconds[13]. With ZODB, on the other hand, the same task takes about half an hour since it is necessary to iterate over every single event and all attachment-capable objects within that event. A custom ZODB index could speed this up, but not without requiring extra code to keep the index synchronized.

2.3.4. Migrations It is common that the database structure changes between versions. Usually, the changes consist of new tables and columns, but sometimes there are more substantial changes. While using ZODB, Indico provided a custom migration script that patched the database directly by iterating over objects and updating them one by one. Not only was this slow, there was also no way to undo such a migration. While it is rarely needed to downgrade in production without restoring a backup, it is more common in development when switching between branches. Having to keep separate databases or restoring backups for this purpose was a waste of developer time and a source of frustration, so a better solution was needed. *Alembic*[14], a library for SQLAlchemy, handles database schema migrations easily. Each revision consists of a Python file with `upgrade` and `downgrade` functions containing the SQLAlchemy logic to be executed when executing or undoing that particular revision. Usually, the code in these functions only executes DDL[15] SQL commands, but any Python code can be used there if needed.

3. The Iceberg

The end-user of a web application only sees the front-end, so they do not get in contact with everything that runs in the back and that actually powers the application. All that users of the CERN Indico instance could notice during the last years is that parts of it became prettier, more functional, and some old bugs disappeared. They were mostly happy with these changes, or became happy upon realizing that the updated UI was better than what they had to use before.

What a regular user cannot see, however, is how massive the changes in Indico have truly been. As can be seen in fig. 1, Indico's codebase has shrunk by 30%, and 85% of its legacy code has been removed.

This reduction in size is not only due to Indico now using third-party libraries where appropriate, but also thanks to the removal of multiple occurrences of very similar snippets of code, which was common in legacy Indico. In addition, ZODB required high amounts of extra code for basic functionality, which is now handled either by PostgreSQL or SQLAlchemy without any custom code.

[12] The largest Indico instance in the world as far as the development team knows
[13] On a consumer PC with a 4 GHz quadcore CPU
[14] `http://alembic.zzzcomputing.com`
[15] Data Definition Language

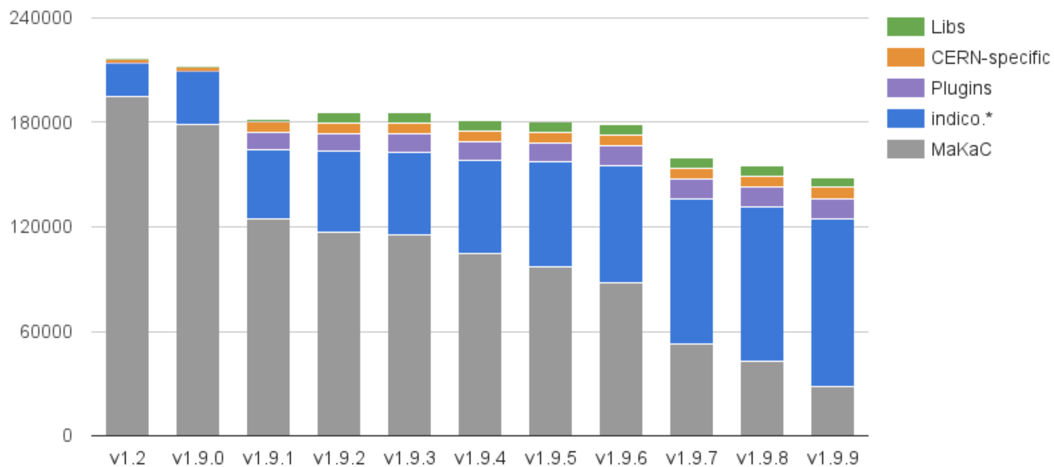

Figure 1. Evolution of the distribution of lines of (Python) code in Indico

4. Conclusion

Icebergs tend to sink ships, but, unlike the Titanic, Indico did not sink and is steaming ahead stronger than ever. With ten successful intermediate 1.9.x releases that cover everything, from room booking, users, or plugins to event timetables or abstract reviewing, the next release, 1.9.10, will already be using only PostgreSQL and no ZODB at all.

To finish the journey towards releasing Indico 2.0 some house-keeping needs to be done to clean-up the remaining code that has not been rewritten yet. Additionally, users who are currently using Indico 1.2 will need a tool to import all the data stored in their existing ZODB database into PostgreSQL.

The biggest project in Indico's history – taking more than three years and around ten man-years – taught the team some very useful things:

- Rewrites are sometimes necessary. Dealing with legacy code is harder than just throwing it away.
- Users rarely like changes. But once they get used to them they usually appreciate them.
- One year of development is not much. The room for error when estimating the effort to rewrite a project as big as Indico[16] is high, which was proven by the fact that it took more than three years in the end, while initially being planned to take only one.
- Building a strong team is essential. Good development practices are important, but team dynamic is even more. This project was only possible thanks to having a strong team with established development processes and the right amount of expertise.

[16] 150k lines of code, initially even more than 200k LOC

References

[1] Ferreira P, Avilés A, Dafflon J, Mönnich A and Trichopoulos I 2015 Indico – the Road to 2.0 *J. Phys.: Conf. Ser. 664 (2015) 052012* doi:10.1088/1742-6596/664/5/052012