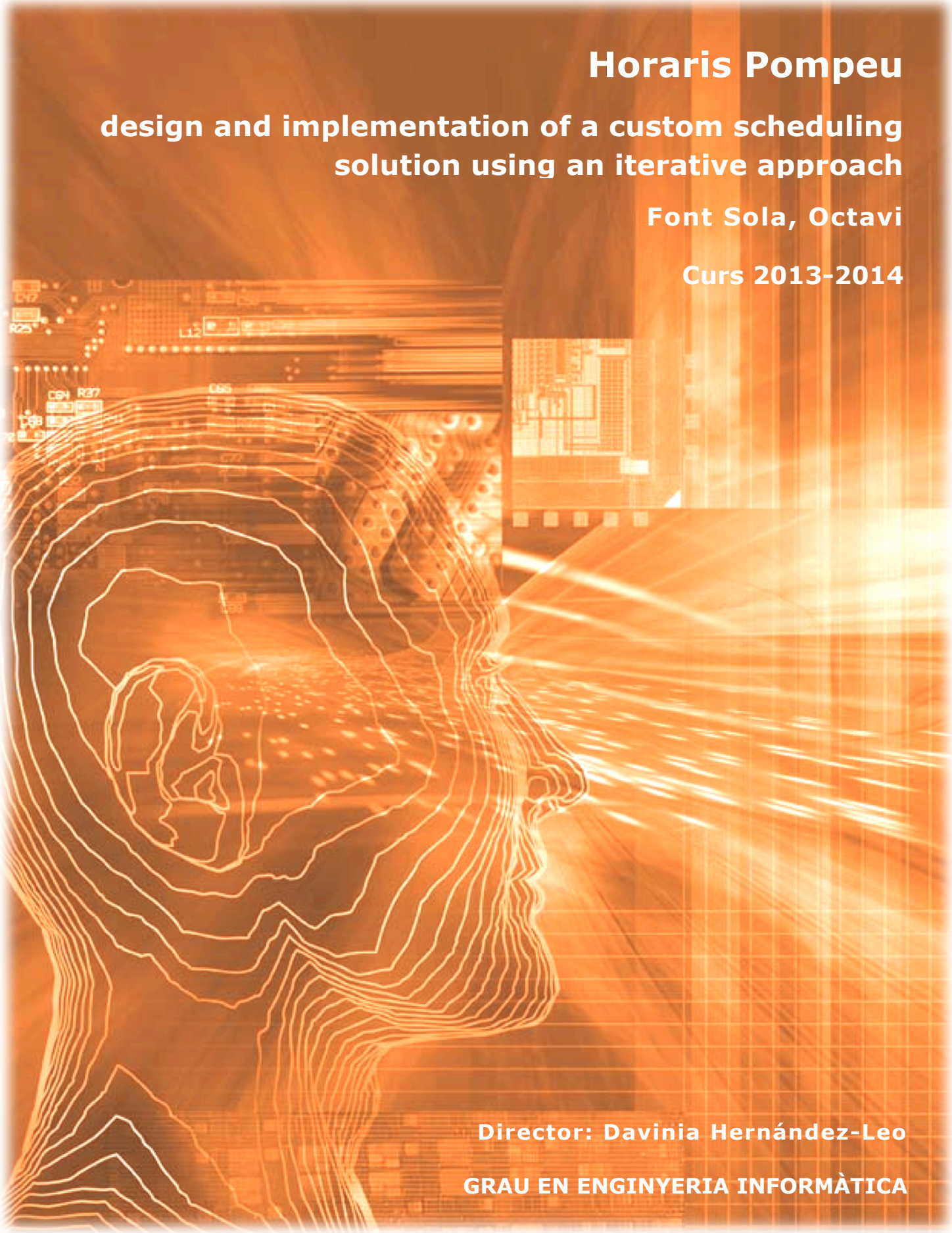
Horaris Pompeu

design and implementation of a custom scheduling solution using an iterative approach

Font Sola, Octavi

Curs 2013-2014



Director: Davinia Hernández-Leo

GRAU EN ENGINYERIA INFORMÀTICA

upf. Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Contents

Abstract

English

This Bachelor's Final Project describes the design and implementation of a platform aimed to create custom schedules for UPF students. *Horaris Pompeu* has followed an iterative driven design, where the requirements and implementation has evolved to best fit its users needs. The report will reflect this development model, presenting motivations and design goals first, followed by an evaluation of each iteration.

As of June 2014, the production system has support for ESUP's 3 degrees: Computer Science, Telematics Engineering and Audiovisual Systems Engineering. Support for the rest of UPF bachelors and masters has already been implemented and will be available next September.

Català

Aquest Treball de fi de grau descriu el disseny i implementació d'una plataforma enfocada a la creació d'horaris personalitzats per estudiants de la UPF. *Horaris Pompeu* ha seguit un procés de disseny iteratiu, on els requeriments i la implementació han evolucionat per adaptar-se millor a les necessitats dels seus usuaris. Aquesta memòria té com a objectiu plasmar aquest model de desenvolupament, presentant en primer lloc les motivacions i objectius inicials, seguides d'una evaluació de cada iteració.

A data de juny de 2014, el sistema en producció té suport per 3 dels graus de la ESUP: informàtica, telemàtica i enginyeria de sistemes audiovisuals. El suport per la resta de graus i màsters de la UPF ja ha estat implementat i estarà disponible aquest proper setembre.

Castellano

Este trabajo de fin de grado describe el diseño e implementación de una plataforma enfocada a la creación de horaris personalizados para estudiantes de la UPF. *Horaris Pompeu* ha seguido un proceso de diseño iterativo, donde los requerimientos i la implementación han evolucionado para adaparse mejor a las necesidades de sus usuarios. Esta memoria tiene como objetivo plasmar este modelo de desarrollo, presentando en primer lugar las motivaciones e objetivos iniciales, seguidos de una evaluación de cada iteración.

A fecha de junio de 2014, el sistema en producción tiene soporte para 3 grados de la ESUP: informática, telemática e ingeniería en sistemas audiovisuales. El soporte para el resto de grados y másters ya ha estado implementado y estará disponible de cara al siguiente septiembre.

Introduction

Motivations

As a student enrolled in two degrees simultanously I've had to deal with a subpar scheduling experience throughout my bachelor. Having to check two or three different timetables was an ordinary nuisance. In fact, at a given time, I had to check 4 different timetables: telecommunications 3rd course, CS 2nd course, optional courses list and audiovisuals 3rd course (for another non-compulsory subject). Not even my weekly habit of writing down my custom timetable into Google Calendar could save me from mistakes, since manual updates are error prone, time consuming and not subject to future changes.

I told myself that something so basic as checking your schedule ought to be simpler, but payed no heed to it until, in March 2013, I took an afternoon to write a lesson parser. Its first version took 4 hours to code and was able to address my needs. In fact, some of the code in production it's still from that first attempt[1].

It took more time than what that first parser led me to believe, but nowadays, *Horaris Pompeu* is a very useful resource for ESUP students. Error free, fast and automatized. The production system is live at www.horarispompeu.com[2].

Methodology

So far as software engineering is concerned, *Horaris Pompeu* has followed an agile approach. In this particular case, where I am both the worst case scenario user and the development team, most software development methodologies would be more of a liability than an asset. As I understand, processes such as waterfall, RUP or SCRUM[3] are ways to establish a framework to facilitate

[1] Https://github.com/octavifs/horarispompeu/commit/c4acc671d4480fe01bc845224201fa80e6c59c5c.
[2] Https://www.horarispompeu.com.
[3] See [Sommerville 2010], [Kruchten 2003], [Sims and Johnson 2012], [Rubin 2012] and [Beck et al. 2001].

fluid communication between the stakeholders and the development team. But with a sole member both driving and implementing the design, communication problems are nil, so the only issue left is correctly framing the problem at hand. Thus, quickly iterating over prototypes is a better approach, and can also benefit from hindsight.

In the following chapter I will define a minimum list of requirements the project will have to meet at completion. From this initial list, I will extract a series of metrics that I will use to measure how far has each iteration progressed. That list of requirements will be also used to abstract the main list of technical challenges this project presents.

After the requirements, metrics and main problems are presented, there will come a series of chapters about each implementation sprint. Each sprint won't take longer than a month and will result in a complete iteration, ready to deploy, of the project. On each implementation chapter we will discuss the changes done to improve our metrics. Any modifications in the initial requirements will be also discussed there.

Chapter 1

Design goals

In this chapter we will introduce the initial requirements that we set to achieve, the problems that those requirements entice and the metrics we will use to measure goal completion.

Requirements

A lot of agile software development techniques gather requirements in the form of user stories. A user story is typically a short sentence where a stakeholder states a need and why fulfilling that need is important to him. There are other ways to gather user requirements (use cases, typically) but I considered that the needs of this project are clear enough that simple user stories are enough to serve this purpose.

In *Horaris Pompeu* case, there are 2 kinds of stakeholders: *STUDENT* and *ADMIN*. In all the following user stories, the *raison d'être* behind every story is to build a product that doesn't get in the way.

- A *STUDENT* can create a schedule from any arbitrary selection of subjects, from any course, from any degree.
- A *STUDENT* can view the custom schedule from any device he owns (desktop, mobile or tablet).
- A *STUDENT* automatically receives updates upon lessons changes.
- An *ADMIN* can parse lessons from the source and save them to a local DB.
- An *ADMIN* can manage update conflicts automatically.
- An *ADMIN* can configure the system to update lessons automatically.

Problems

From the previous requirements list, it is possible to divide the implementation tasks in 5 main problems:

- **Presentation**: Presenting choices in a way that doesn't overwhelm the user is key to make the calendar creation process as fast and easy as possible.
- **Ubiquity**: What is the best way to display a calendar in multiple platforms with minimal configuration and seamless synchronization?
- **Parsing**: The source data for the schedules is semi structured, since the input process has many manual steps involved. That will condition the parser and the database schema.
- **Merging**: Related with the previous problem. If manual changes are needed in the local DB (to fix errors in the parsing), how can we later synchronize our local DB with the timetables source, when those change?
- **Automation**: Can the parsing and merging process be automated? How do we deal with backups? How do we inform of errors?

Metrics

For each problem presented in the previous section, we have a metric to measure our level of progress achieved:

- **Presentation**: Percentage of calendars created per session. Total number of calendars created.
- **Ubiquity**: Number of platforms supported. Number of subscribed calendars.
- **Parsing**: Accuracy. How many entries have to be manually changed?
- **Merging**: Accuracy. Are there any errors or duplications while updating an entry that had been manually changed?
- **Automation**: How much admin time does *Horaris Pompeu* require to maintain a good QoS (no complains from users, no downtime, no errors in the calendars)?

Chapter 2

Legacy system challenges

In this chapter we will study the current implementation of the scheduling system in depth in order to analyze which difficulties will have to be faced when adapting the new solution to the old framework.

Why is an adaptation needed?

Horaris Pompeu does not intend to supersede the current scheduling system, since this is a political decision which I hold no power over. Instead, this product is built to work alongside the official solution, using its data in a more user-friendly manner.

Since it will be necessary to parse the data from the current system before we can use it, it is very important to understand how the actual system works and what kind of problems will arise in the implementation of the parser.

Legacy system description

Currently, the timetables for ESUP are stored in static HTML files. Each degree, course and group has 1 file. Each course and term has a list of subjects and its schedule can be retrieved from the correct static HTML file.

Those HTML files are (semi) manually updated by the ESUP secretary department and published to the web, in this URL: http://www.upf.edu/esup/docencia/horaris1314/.

Subject lessons are stored in one or multiple timetables (subjects can be shared between degrees). Each timetable is a static file with a set of tables, where each one shows the schedule for the week.

Grau en Enginyeria en Sistemes Audiovisuals

Primer curs - Primer trimestre - Grup 2

Setmana 1: Del 23/09/2013 al 27/09/2013
Setmana 2: Del 30/09/2013 al 04/10/2013
Setmana 3: Del 07/10/2013 al 11/10/2013
Setmana 4: Del 14/10/2013 al 18/10/2013
Setmana 5: Del 21/10/2013 al 25/10/2013
Setmana 6: Del 28/10/2013 al 01/11/2013
Setmana 7: Del 04/11/2013 al 08/11/2013
Setmana 8: Del 11/11/2013 al 15/11/2013
Setmana 9: Del 18/11/2013 al 22/11/2013
Setmana 10: Del 25/11/2013 al 29/11/2013
Setmana 11: Del 02/12/2013 al 06/12/2013

Setmana 1	DILLUNS	DIMARTS	DIMECRES	DIJOUS	DIVENDRES
Hora	23/09/2013	24/09/2013	25/09/2013	26/09/2013	27/09/2013
08:30-10:30	FESTIU	FESTIU	Algebra Lineal i Matemàtica Discreta **TEORIA** T2A: 52.221 T2B: 52.119	Enginyeria d'Interacció **PRÀCTIQUES** 8:30 - 9:30 P201: 54.005 9:30 - 10:30 P202: 54.005	Xarxes i Serveis **TEORIA** Aula: 52.019
10:30-12:30	FESTIU	FESTIU	Introducció a les TIC **TEORIA** T2: 52.019	Càlcul i Mètodes Numèrics **SEMINARI** 10:30 - 11:30 S201: 52.105 S203: 52.321 S205: 52.429 11:30 - 12:30 S202: 52.105 S204: 52.321 S206: 52.429	Algebra Lineal i Matemàtica Discreta **PRÀCTIQUES** P201: 52.119 P202: 52.221 P203: 52.019
12:30-14:30	FESTIU	FESTIU	Enginyeria d'Interacció **TEORIA** Aula: 52.221	Xarxes i Serveis **TEORIA** Aula: 52.019	Introducció a les TIC **TEORIA** Aula: 52.023

Figure 2.1: Example of a legacy timetable

Some of the biggest shortcomings of the system are:

- the formatting is not consistent. Lessons are typically presented following a certain pattern, but this is not always the case. Since data is manually entered, there tend to be different styles in date formatting, groups, classrooms, etc. The differences are subtle but make the data hard to parse.
- poor support for redefining time. If the cell time is not correct, another time for the lesson has to be specified inside the cell.
- poor support for multiple lessons. If different subjects take place in the same time slot, they are separated by dashes (—-).
- information duplicity. Since lots of subjects are shared between degrees (but not groups!), a lot of the lessons found in a timetable are the sames ones found in another timetable.
- because of the rigid structure in the timetables, some students have to check many of them to get a full schedule for all their lessons, especially the ones who study multiple degrees or have subjects pending from previous courses.
- no mobile support of any kind. It's especially difficult to navigate the ESUP page to the timetables, given it uses a dropdown on the sidebar.

Another important thing to consider is the frequency in which the schedule is updated. It is very typical in the ESUP's case that a schedule will have to be updated various times throughout the term.

Legacy system challenges

All the points above set us a series of challenges regarding parsing and automation mainly.

- It will be difficult to normalize the data, starting from a denormalized set (as the sources is).
- Inconsistencies in the format will make parsing error prone. The parser will have to be aware of its inability to process the data and somehow inform the admin that some lessons need manual editing.
- If manual editing is needed, there has to be a way to perform this kind of administrative tasks easily, instead of having to deal with the database directly, which would be slower and more error prone.
- Updates in the schedules force us to keep track of insertions and deletions in the sources. This is especially hard if the parser is not good enough and some of the parsing has to be done manually. If we are not careful, this may cause synchronization problems between the sources and the new platform, with some updated being missed by the system, or some entries duplicated.

- Since updates may go wrong, they should be easy to roll back.

Chapter 3

Sprint 0 - Initial prototype

This will cover the initial sprint of the project, from *21-03-2013* until *28-03-2013*. Tagged as v0[1] on the *Horaris Pompeu* GitHub repository.

In this initial series of sprint chapters, I will talk about the initial decisions regarding the chosen technology and architecture of the service. Afterwards, I will proceed to explain the progress achieved in this iteration based on the criteria established in the *design goals* chapter. This will cover the initial parser and schema design, along with the application views and management commands.

Technology stack

Horaris Pompeu needs are those of a typical web application. A persistence layer (database) to store the data (subjects, lessons, calendars, etc.) and views to display such data in a useful manner. If anything, the main difference between this project and other web applications will be the sources where data is obtained from and the fact that the viewing of the calendars will not be done on-site (more about this later).

There are many frameworks and libraries available to aid the development of systems such as this. Most of these frameworks are inspired in the MVC design pattern[2]. They use this strategy to ease the separation of concerns between components and favor loose coupling. In short, MVC stands for Model View Controller. **Models** represent data, and tend to be backed by a persistency layer. The models role is to describe the data schema of the application and notify the views and controllers when its data changes. **Views** request information from

[1] Https://github.com/octavifs/horarispompeu/tree/v0.

[2] See [Krasner and Pope 1988] for an introduction to the MVC pattern and [Leff and Rayfield 2001] to know more about its usage in web applications.

the model, in order to present the data to the user. **Controllers** are in charge to manipulate the models.

The MVC paradigm was born in the 70's, as a software design pattern to help in the development of graphical user interfaces. In the web development world, MVC started to gain traction with the release of the *Spring* framework (*Java*, 2003) and *Rails* (*Ruby*, 2004). Those kinds of frameworks have become really popular and right now most programming languages have at least one major mature alternative, so choosing between them may be a difficult decision.

In the case concerning this project, since I wanted speed of iteration to be one of its distinguishing factors, I preferred to search for a framework in a language I was already familiar with and had a wide range of libraries (for tasks such as parsing HTML). After a period of deliberation, I decided on *Python*. *Python* is dynamic and interpreted language, born in the 90's, known for its terse syntax and focus on readability. *Python* is widely used in a lot of fields, such as scientific computing, system administration and web development. I had worked with the language before, and already felt comfortable with it. Moreover, I knew there was a reasonably large web development community invested in the language and that development tools and documentation would be readily available.

Once decided on the language, I had to choose the framework. There happen to be many open source frameworks for web development in *Python*. Among the ones I considered, we could find *Django*, *Flask*, *Pylons*, *Web2Py*, *Pyramid*, etc. Some of them are more feature complete, some others take a more minimalistic approach, but at the end I chose *Django*[3]. *Django* is the leading web framework in the *Python* community. A direct reply to *Ruby on Rails*. Among the breadth of features it offers, I particulary liked the ORM (a mapper between object models and the actual database tables and queries needed to access them), template system and admin site. The admin site was especially interesting (and unique to *Django*), since it could be used to manually input data if there was any need for corrections. In the previous chapter we have already discussed the number of inconsistencies found in the original sources so, expecting problems in the parser stage, I thought it wise to find a way to cheaply fix the small amount of mistakes there may ocurr manually.

Finally, there was a choice to make regarding the database backend. *Django* has native support for *PostgreSQL*, *MySQL*, *SQLite* and *Oracle* backends. I ended choosing *SQLite*, for testing and production purposes. I considered that in this particular case, since writes to the database will mostly come from batch processes (no users) and the number of concurrent users in the site is not expected to be high, it was preferable to use the simpler situation. In this case, *SQLite* has been perfectly reliable, very easy to set up and even simpler to backup.

[3] Https://www.djangoproject.com/.

Architecture

Django favors a modular approach when creating a new project. A typical good designed *Django* project will be composed of multiple apps, each with its own distinct purpose. *Django* also has a 3rd party app system, which meshes naturally with the custom created apps and makes it easy to share and reuse modules of 3rd parties (increased speed of iteration).

In our case, since all we want has to do with calendars (parse, store, create and show them) I decided that it made sense to develop the project as a single *Django* app and a few 3rd party add-ons.

In the following diagram, I've drawn the file structure of the project in it's initial iteration. The last iterations still follows most of this structure.

```
/
    - README.md  # general information about the project. Installation instructions.
    - horarispompeu/
        - settings.py  # general settings of the Django project
    - timetable/  # Django app
        - admin.py  # Describes models representation in the admin interface
        - models.py  # Describes the models
        - urls.py  # Describes the url routing
        - views.py  # Describes the business logic of the app
        - templates/  # Holds the HTML templates for the app views
        - static/  # Static files needed by the templates
        - management/commands/  # Holds commands that will be run as batch operations
```

Presentation

Most efforts in the presentation aspect were directed towards engineering the views and flow between them.*Horaris Pompeu* must serve as a fast selector of subjects for any student, so we have to construct an interface that makes it easy to choose subjects.

Since it is important not to overwhelm users by offering too many choices (a list of all the subjects, for example), I chose to make this subject selector using a tiered approach, very similar to how the timetables are organized in the ESUP page. The flow would be as follows:

1. Select degree(s)
2. Select course(s) and group(s)
3. Select subject(s)
4. Get calendar

To this basic flow, I also added a landing page, that explains the purpose of the application in a few sentences.

What we have then, is a series of forms linked together, where the data sent to the previous form determines the data shown in the next one. Displaying choices in this tree-like structure minimizes the number of valid branches.

URL flow, as defined in `urls.py` and `views.py`:

1. `/`
2. `/grau`
3. `/curs`
4. `/assignatures`
5. `/calendari`

Once reached the `/assignatures` view, we can finally collect the list of subjects the student is taking, and from this we create a custom calendar, choosing the corresponding lessons.

At this point in the implementation, the 3 stage subject chooser was already implemented, but the necessary templates to display the data were not finished, and the calendar creation did not work, so there was still much to do before this release could be considered apt for production usage.

Ubiquity

In the design goals chapter I've talked that one of the main objectives of the project is to make the schedules accessible from any kind of device. Achieving this may be really hard if you start to develop a custom solution for every platform, since this would at least encompass apps for iOS, Android and a Desktop version of the timetable. Instead of focusing on developing a custom solution to display the timetables though, I explored what kind of options were already available in the market. Calendars are basically a solved problem, and there are established protocols and platforms which offer easy interoperation between them.

The first option that I considered was using the Google Calendar API. A very high percentage of people use GMail as their main email account, all UPF students have a Google Apps account (with Calendar) and synchronization between mobile, tablets and desktop is trivial using this platform, on Android and iOS.

The Google Calendar API[4] allows creation of calendars, addition and deletion of events, permissions management (sharing calendars with others), etc. The API is free to use, as long as you perform less than 100.000 daily API calls (each

[4] Https://developers.google.com/google-apps/calendar/.

REST request is considered an API call). This API can be used in your own accounts or on behalf of others, using the OAuth authentication framework.

Using the API was very tempting, as it would have allowed *Horaris Pompeu* to add calendars to each student's own account with a few clicks and no security concerns. It would also make it possible to update those same calendars, as long as the user wouldn't revoke *Horaris Pompeu*'s permissions to do so. Still, there were some concerns about how the API usage limits may had affected the real world usage of the project.

To further test the feasibility of using Google's API, I ran some calculations on the limits it would force onto the project. In the following section I will detail the estimate amount of calls needed to create a calendar:

```
Create a calendar:
10 weeks/term * 5 subjects/week * 3 lessons/subject = 150 lessons/term
100000 calls/day / 150 lessons/term = 666 calendars/day
```

666 calendars a day may seem like a fair figure, but this is a best case scenario estimate. We are creating calendars per term (whole year calendars would diminish the calendar creation rate by a factor of 3) and the restrictive API usage makes problems such as updating harder, since deleting a calendar and writing it form scratch is not cost effective. Updating under the Google Calendar API should be something more fine grained. Also, creating a calendar remotely would be slower than doing so locally, so maybe an asynchronous task queue, that performed calendar creation in the background, would be needed to avoid stalling user requests (complicates architecture, requires more resources and setup).

Overall, even though the API makes the process of creating new calendars very simple for the user, it places a lot of restrictions on the implementation side, complicates the project's architecture and may stunt its growth and long term viability.

Since I found the API route too restrictive, I searched for more options that struck a better balance between my ability to control the platform and ease of use by the students. What I discovered was a standard file format for event exchanges between applications. This format, called iCalendar, was created by the IETF (RFC5545 [B. Desruisseaux and Oracle 2009], RFC5546 [C. Daboo and Apple Inc. 2009] and RFC6868 [C. Daboo and Apple Inc. 2013] and is well supported by most calendar applications (including Google Calendar). iCalendar files can be imported or subscribed from a URL. This last option is especially interesting in our case, since it makes possible to store the calendars in the server, while the calendar app of the client regularly checks the url for updates. This means that we only have to make sure our calendar files are updated and available with the parsed data, and the user client (Google Calendar, for example) will take care of the rest. This option is also especially interesting because the

client caches the calendar locally, so even if *Horaris Pompeu* is down, we will still be giving service to our current users.

iCalendar subscription is supported by Google Calendar, but the feature is rather hidden in the user interface, and only available from the desktop version of the site (API or mobile version don't have that option), so the subscription flow is not as good as it was with the API, but the benefits from an implementation standpoint weighted more.

To create icalendar files in Python I've used a 3rd party icalendar library[5]. With this library converting the custom representation of the data as stored in the DB to `.ics` files is very straightforward. In fact, this initial version already has[6] a function that writes lessons to icalendar files.

Parsing

The parsing problem encompasses 2 steps. The first one is to structure and scrap the data from the legacy timetables. The second one is how to organize this data in a database schema, so that it stays true to the relations it represents and makes it easy to diff updates.

parser

The parser was placed initially on `/timetable/management/commands/_parser.py`[7].

The parsing of the legacy HTML timetables is done in 2 phases. The first one obtains the data stored in each cell along its context (date and time). The second one processes the cell data into one or various distinct lessons.

To parse the HTML I've used BeautifulSoup[8]. This is a well established *Python* library for HTML parsing, that makes it very easy to traverse an HTML tree. Since this is a mature project, it is able to process even broken HTML with very good results, so it will be more than enough for the project needs.

During the HTML parsing, I use 3 nested loops. The first one loops through every table (week) of the HTML. The second one iterates through every row of the table (time slots). In this second iteration, the date (column header) and time (row header) is saved in its own variable. When accessing the cell contents (in a 3rd nested loop), the date and time values previously saved are used to preserve the context of the lesson(s). The third loop rolls over each cell of each row, extracts its content and tries to parse it.

[5] Http://icalendar.readthedocs.org/en/latest/.
[6] Https://github.com/octavifs/horarispompeu/blob/v0/timetable/management/commands/_parser.py#L104-L121.
[7] Https://github.com/octavifs/horarispompeu/blob/v0/timetable/management/commands/_parser.py.
[8] Http://www.crummy.com/software/BeautifulSoup/.

Once the text from the cell is obtained, `parselesson` function is launched. `parselesson` takes the text from the cell, the date and hour and returns a list of lessons contained inside. In broad strokes, the function proceeds like this:

1. strip the cell for blank lines
2. parse the subject
3. parse lesson kind (theory, practice, seminar, etc.)
4. parse specific hour (if any)
5. parse groups (if hour is found, go to #4)
6. store lesson (if separator is found, go to #2)

It is important to note the difficulty to correctly parse some fields, such as the subgroups (specific seminar, practice or theory groups, usually Sxxx or Pxxx).

> **Some subgroup format variants:**
> Sxxx: classroom
> Sxxx i Syyy: classroom
> Sxxx, Syyy, Szzz: classroom
> Aula: classroom

Each lesson returned by `parselesson` would have the following fields:

- subject
- kind
- group
- room
- date_start
- date_end
- raw_data

`raw_data` is a special field I added, to store the original text of the cell, just so it would be easier to detect and correct errors during the parsing.

models

All models are defined in timetables/models.py[9].

Faculty : To allow the usage of this schema in multiple faculties, I added this model, even though it has only been used to hold one faculty (ESUP).

AcademicYear : To allow the usage of this schema yearly without having to flush the previous database, I also store the academic year as an independent entry.

[9] Https://github.com/octavifs/horarispompeu/blob/v0/timetable/models.py.

Degree : Each degree has a name and is linked to a faculty.

Subject : Each subject has a name and is linked to a faculty. In ESUP, subjects may be shared among degrees and only depend on the faculty.

SubjectDuplicate : Created to deal with some problems with the *subjectparser* command. The sources used in *subjectparser* had some subject duplication so, to make sure re-running the script does not re-enter unwanted subjects, subjects deleted from the Subject model are stored in this one.

SubjectAlias : Saves all the aliases and typos used in the legacy timetables to refer to a specific subject.

Class (later renamed to Lesson) : Has all the information on the subject taught, group, subgroup, kind of lesson, classroom and time of start and end.

DegreeSubject : Relates Subjects to Degrees. Also adds information on which term, bachelor year, academic year and group the subject is taught.

Calendar : A simple model. Stores references to iCalendar files and all the subjects it includes.

Automation

In this initial iteration I created a few batch commands necessary to synchronize the local database with the legacy calendars and started configuring the admin interface for manual management of the system.

Django has support to create scripts that can interact with the app models and framework functions. This feature is very useful in our case, where we have to run a series of batch operations (parsing legacy calendars and updating database and our custom calendars accordingly). In this case, I've created 3 commands:

- management/commands/subjectparser.py[10]
- management/commands/aliasparser.py[11]
- management/commands/classparser.py[12]

subjectparser.py

One of the problems when parsing the legacy timetables is that the subject name is not consistent (some subjects have various aliases, or typos). To obtain a reliable list of subjects, I used the official UPF syllabus from each degree:

[10] Https://github.com/octavifs/horarispompeu/blob/v0/timetable/management/commands/subjectparser.py.
[11] Https://github.com/octavifs/horarispompeu/blob/v0/timetable/management/commands/aliasparser.py.
[12] Https://github.com/octavifs/horarispompeu/blob/v0/timetable/management/commands/classparser.py.

- Bachelor's degree in Computer Sciences[13]
- Bachelor's degree in Telematics Engineering[14]
- Bachelor's degree in Audiovisual Systems Engineering[15]

This script parses the page, obtains the subject names and saves them into the databse, if the entry did not previously exist.

aliasparser.py

As mentioned in the previous section, parsed subject may have different aliases. This runs the parser on all legacy timetables and inserts any non-registered alias in the database. Then, those aliases can be manually linked to the corresponding subject, so when we are inserting the lessons (3rd script), they have the correct subject.

classparser.py

This is the script that runs the parser and saves the lessons to the database. Its implementation only saves the lessons to the database and little else. Does not deal with updates or deletions of any kind.

[13] Http://www.upf.edu/pra/3377.
[14] Http://www.upf.edu/pra/3376.
[15] Http://www.upf.edu/pra/3375.

Chapter 4

Sprint 1 - Production ready prototype

This will cover the *first* sprint of the project, from *04-04-2013* until *28-04-2013*. Tagged as v1[1] on the*Horaris Pompeu*GitHub repository. Diff[2] between v0 and v1.

v1 was the first iteration that I considered feature complete. It ran in production during the 3rd term of the previous academic year (2012-13) and was beta-tested by a handful of users.

In this iteration I finished the views and templates, the creation of calendars, tweaked the database schema, improved the parser and refactored the management scripts and admin interface, for easier administration of the site.

Licensing

Since the beginning, I wanted *Horaris Pompeu* to be an open source project. The code was published in a public GitHub repository[3] from the very start, but I had not licensed the code properly. Even though GitHub's terms of service specify that public repositories must be open source, no license is specified, so my project was in a legal void regarding licensing.

My objectives behind the licensing of the software were more concerned towards preserving credit for the work done, trademarks and avoiding possible liabilities caused to 3rd parties. Ensuring that the code would continue to be free, or that

[1] Https://github.com/octavifs/horarispompeu/tree/v1.
[2] Https://github.com/octavifs/horarispompeu/compare/v0...v1.
[3] Https://github.com/octavifs/horarispompeu.

further modifications to it had to be published under the same terms was not a concern of mine.

For all these reasons, I chose to license the code under the Apache v2 License[4]. To license an open source project the main requirement is to upload a LICENSE file, with the contents of the chosen license. It is also recommended to put a boilerplate notice on top of every source file, referring to the full LICENSE[5].

Presentation

The business logic on the views mostly remained the same, but I had to implement the calendar creation view.

On the calendar creation view, the list of selected subjects is retrieved and sorted by primary key. This sorted list of subjects is later hashed (using SHA1 [Information and Standards]). The hash is performed as a way to make sure iCalendar files will be unique. Even though the number of possible subject combinations is very high, SHA1 is a proven and widely used cryptographic hash, with a collision rate on the order of 10^-45, which makes name collisions very unlikely.

Once the hash is performed, I query the Lessons for that calendar and, with those Lessons, I create a new iCalendar, which is saved to a file into a static folder. A reference to both the subjects and the calendar file is stored into a Calendar model object.

Saving calendars this way (with a hash, to check for collisions) makes it trivial to check if that calendar had already been created previously (and avoid recreating it, if that is the case) and is also a good way to randomize file names. If, for example, calendar names were created sequentially (1, 2, 3...) any person could retrieve all the calendars stored in the database, which is not a problem per se, but may be undesirable.

Apart from finishing the business logic, I also wrote the templates for the views. I followed a simple design, where every template shown a single column of multiple choice options. Once submitted, a new view with more choices was presented, until the calendar was created. Once the calendar was done, a view with a URL to the calendar and some instructions as to how to proceed was displayed.

This first version of the templates was inspired by the UPF visual identity[6] and tried to preserve the iconic whites and reds. The same pantone is used (pantone 186) and typography tries to stay close to the official *Excelsior*, opting for Georgia, a widely available and web safe font.

[4] Http://www.apache.org/licenses/LICENSE-2.0.html.

[5] Literature regarding software licensing can be found on [Github 2014], [Open Source Initiative 1988], [Foundation 2004], [Free Software Foundation 1991] and [Free Software Foundation 2007].

[6] Http://www.upf.edu/marca/evolucio/.

Figure 4.1: Design of iteration v1 of *Horaris Pompeu*

Ubiquity

Each iCalendar file was given a title with the url of the site (www.horarispompeu.com). I considered this useful to remind users where they got the schedule, since once it has been generated and subscribed, there is no longer any need to access the page and people may forget about the project. A basic move to raise brand awareness.

I also had to add timezone information in the file, so Google Calendar displayed dates correctly.

Apart from these tweaks in the code, I also had to change some settings in the production server to modify the headers used when serving the files.

```
**NGINX .ics config**
add_header Content-Type "text/calendar;charset=utf-8";
charset utf-8;
```

If mimetype and charset was not specified, Google Calendar would not load the URL, or would read the charset wrongly and display accents and other special characters as garbage.

Parsing

Minor changes in that regard. As I already mentioned in the previous chapter, I renamed the *Class* class to *Lesson*, since the choice of words had been really unfortunate and confusing that first time. Also, a `__hash__` method was added to the *Lesson* class, so objects could be compared by content equality now (and not mere reference). This enabled the usage of the `==` operator and the usage of sets to store *Lesson* objects.

Merging

The previous *classparser* command was renamed to *lessonparser* and rewritten from scratch. In the previous version, the command was only able to add new lessons to the database, but didn't have any sort of conflict resolution mechanisms. In this new version, some improvements were made:

- When downloading the legacy timetables, encoding is specified and correctly set.
- Last legacy timetables update were saved, so the first step of the update is to check whether the file has changed at all. If it has not, skip timetable.
- If changes in the timetable have taken place, both the current and the previous stored timetable are parsed. Those parsed lessons are put on a set and diffed. With this, we get the subset of inserted and deleted lessons.
- Insertions and deletions are performed in the database.
- Calendars are updated.

This new system, while much better than the previous one, still had problems. Basically, if any manual changes were done in the database lessons, and some of those manually modified lessons had to be deleted, they may not get picked up by the delete command, so the possibility of having rotted entries if the database was manually managed was real.

In real use though, the problems were not really noticeable, but this is also because the number of real users of *Horaris Pompeu* at that time was probably no larger than 5 people and the calendars we used happened to be using one of the less conflictive set of lessons.

Automation

`admin.py` was heavily reworked, so that every model in the database had useful views and filters. Most of the work was done to make sure Lessons and Subjects were sortable and that it was easy to discover whether any of those had any

missing information. The new filtering mechanisms would also be useful when trying to manually a specific lesson.

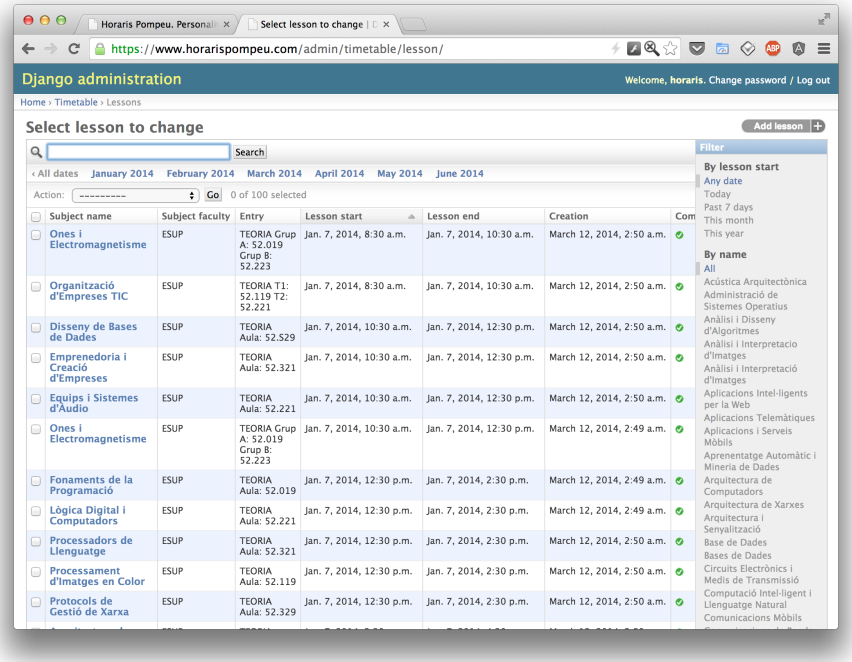


Figure 4.2: *Horaris Pompeu* admin interface. List view of Lesson objects

Also, more models were introduced to ensure non-destructive Lesson updates. The new system added the following models:

LessonArchive : Every time a lesson is deleted, store it in the LessonArchive table. With this, the information is not destroyed, so if there has been an error, it's easier to recover the data.

LessonDelete : Every time a lesson is deleted, a referenced to the newly created LessonArchive is saved, along with a timestamp.

LessonInsert : Every time a lesson is inserted, a reference to the newly created Lesson is saved, along with a timestamp.

All this triggers described in the models are enforced via *Django* signals, a functionality that enables developers to launch certain actions when certain events take place in the database (inserts, deletes and updates).

Chapter 5

Sprint 2 - Mobile compatible responsive design

This will cover the *second* sprint of the project, from *20-08-2013* until *25-09-2013*. Tagged as v2[1] on the *Horaris Pompeu* GitHub repository. Diff[2] between v1 and v2.

v2 centered around improving the templates by adding mobile support to the website, streamline the subscription process, better documentation regarding the deployment process of the project and some changes in the management commands to improve the management of the site.

Presentation

The major visible change of this new iteration was the rewrite of the templates. The number one complain by the few beta testers was that v1 had no mobile support, so using it in mobile devices was inconvenient.

In order to solve that, I decided to use a front-end framework with built-in support for developing responsive websites that automatically reflow themselves depending on the device used.

Front-end frameworks have recently come into the spotlight, as a result of the increasing number of platforms and browsers a website has to optimized for. In my case, I used Bootstrap[3], an open source project started at Twitter. Bootstrap

[1] Https://github.com/octavifs/horarispompeu/tree/v2.
[2] Https://github.com/octavifs/horarispompeu/compare/v1...v2.
[3] Http://getbootstrap.com/.

offers good support to define responsive layouts and many default styles for a wide range of HTML elements. This makes it easy to prototype attractive designs in little time and also ensures that those designs will be compatible accross browsers (Bootstrap default styles take care of that).

All the templates were rewritten to make use of the Bootstrap framework, but the core idea remained. The subject selector still was a list of checkable items, and the main chromatic theme was dominated by white and red.

A few new elements were introduced: a navigation bar, that displayed the step in the creation of the calendar and a background image. The idea behind the addition of the background image was to take advantage of all that blank space in the background and give a litle bit of character to the website, in a similar fashion to what wall calendars do.

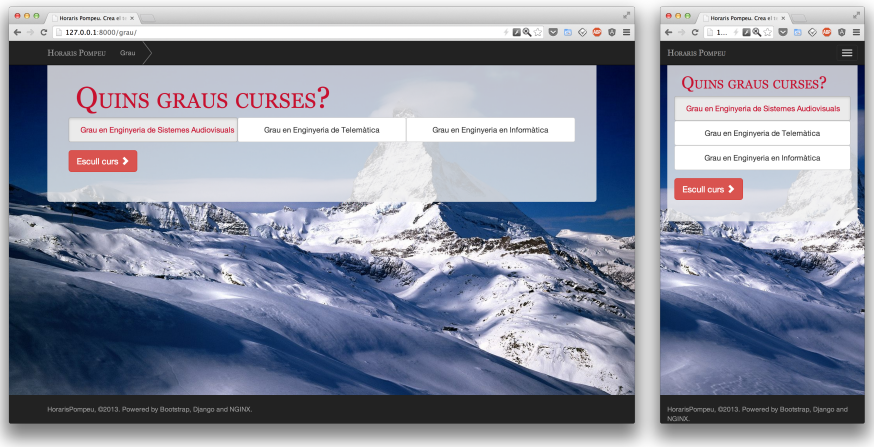


Figure 5.1: With the new responsive design the number of columns shown depends on the width of the browser window.

Ubiquity

One of the other major sources of complaints from the initial beta testing of the website was that the process of subscribing to Google Calendar was not clear enough. As I explained previously, I decided to abandon the usage of the API in favour of using iCalendar files, even with the problems of having to deal with an inferior subscription workflow. Also, since I added explicit support for mobile devices, and the mobile version of google calendar does not offer the option to subscribe to `.ics` files, I decided I had to offer a working alternative to those users, so they could complete the process from within their own device.

Since no API is available (current or deprecated versions) to add iCalendar subscriptions to a Google Calendar account I had to write my own alternative. To do this, I reversed engineered the commands issued by the google calendar web application using the Chrome developer tools.

Using the network tab on the developer tools I inspected the flow from login until adding a calendar:

```
GET https://www.google.com/calendar/
GET (redirect) https://accounts.google.com/
ServiceLogin?service=cl&passive=1209600&continue=https://www.google.com/
calendar/render&followup=https://www.google.com/calendar/render&scc=1
POST https://accounts.google.com/ServiceLoginAuth
    {
        GALX:tjHPwrFAGZY
        continue:https://www.google.com/calendar/render
        followup:https://www.google.com/calendar/render
        service:cl
        scc:1
        bgresponse:!AOLBAIT9oRsN60Ra7G6io6QSvgIAAAAlUgAAAAYqAQKVzBdtaHnLkz8
        88MhNsMXYJfRvkgJqzEhDY93lCl0yLtZFUM0y8R4Ye8aHAE4MqtqvMHuaB--p8k-Zpg
        c_OioEWMcAakWHarwjex3tfHbmEDjWncgy590W2yoi9W8RjByFJlJKanuMQeNSBS8gd
        YnXjUGH4WxFvfReLOesVTKCMON7z19J3Uo_boXMyO6GvswQO55AR-
        YCSOzQHUjGI5RA3Wj2iSOjHa1ZendiTyV1HZK4vhltIlFvwDOpvvx3GL_-Se-
        JFObxOFlYHumRVQO-IWvmiOb2ZXiVYMNMzEsyhOgWHsGNDTxNONxl-
        bB49ltL3LqJaf_HAGTppFUHt4u92RU
        pstMsg:1
        dnConn:
        checkConnection:youtube:119:1
        checkedDomains:youtube
        Email:your_email@gmail.com
        Passwd:your_password
        signIn:Inicia la sessió
        rmShown:1
    }
GET (redirect) https://www.google.com/calendar/render
POST https://www.google.com/calendar/addcalendarfromurl
    {
        curl:https://www.horarispompeu.com/example.ics
        cimp:true
        cpub:false
        secid:ceHOp6Iqix4jfHwVgKf84xzHOFA
    }
```

The flow exposed here are the main requests involved in the process. For a complete log download the HAR file of the experiment (source). You may use

an online viewer such as chromeHar[4].

Apart from this flow, a lot of cookies are setup during the process. Some of them are set as part of the flow mentioned above and some others are set as part of other AJAX calls executed when drawing the pages.

```
Cookies:
APISID, CAL, HSID, NID, OGP, OGPC, OL_SESSION, PREF, SAPISID, SID, SSID,
T, secid
```

The problem in this case is to discover which cookies actually need to be setup so that out calls to `POST https://www.google.com/calendar/addcalendarfromurl` will succeed.

Since Google Calendar is a complex page and javascript execution seems necessary to correctly setup all the cookies, I searched for fully featured HTTP clients, that worked more like browsers rather than a typical headless client like `curl`. What I found was a project named PhantomJS[5], which is a headless browser, based on WebKit, that can be operated programatically through a Javascript API. Using this headless browser I programmed a script (`casperjs/addICSCal.js`[6]) which simulated a user going through all the steps I described above. This script takes an email, password and iCalendar URL from the command line and is able to log in and subscribe you to that URL automatically.

Even though the script made it possible to add automatic subscriptions, this feature introduced security concerns to the application. Since the script works by logging itself as the user, we have to gather the GMail password of our users if they want to use this. This is a shady practice and could perfectly be used to steal identities (not the case, but it is a legitimate concern). In fact, even though the password was not saved in the system at any point, it was still possible than a man-in-the-middle attack could capture the password, since it was sent as cleartext (the site was HTTP only).

Given this grave security concern, were this feature published as the production server was configured, *Horaris Pompeu* users would be vulnerable to identity theft (and possibly worse things) so, to prevent that, I also added support for HTTPS throughout the whole site. Later in this chapter I will talk in more detail about the production system of *Horaris Pompeu*.

Apart from creating this script, I had to add a new view (`/subscriu`). This view only listens to `POST` requests, with 3 parameters:

- calendar url
- email

[4] Http://ericduran.github.io/chromeHAR/.
[5] Http://phantomjs.org/.
[6] Https://github.com/octavifs/horarispompeu/blob/6ada413fdcf8f2e3beadfdd41c2256f2711736ab/casperjs/addICSCal.js.

- password

Each time the view is invoked the `casperjs/addICSCal.js` script is called and an automatic subscription is performed. The view does not have any specific template, since it's triggered via AJAX from inside the `calendar.html` template.

Overall, the new feature was convenient from a usability standpoint but controversial as far as security is concerned, and even though by adding HTTPS all man-in-the-middle attacks are greatly mitigated, it could still be possible that a vulnerability in the server may be used to change the service configuration and store the passwords somewhere. As far as the logs are concerned, the POST body (with password and email) are never stored.

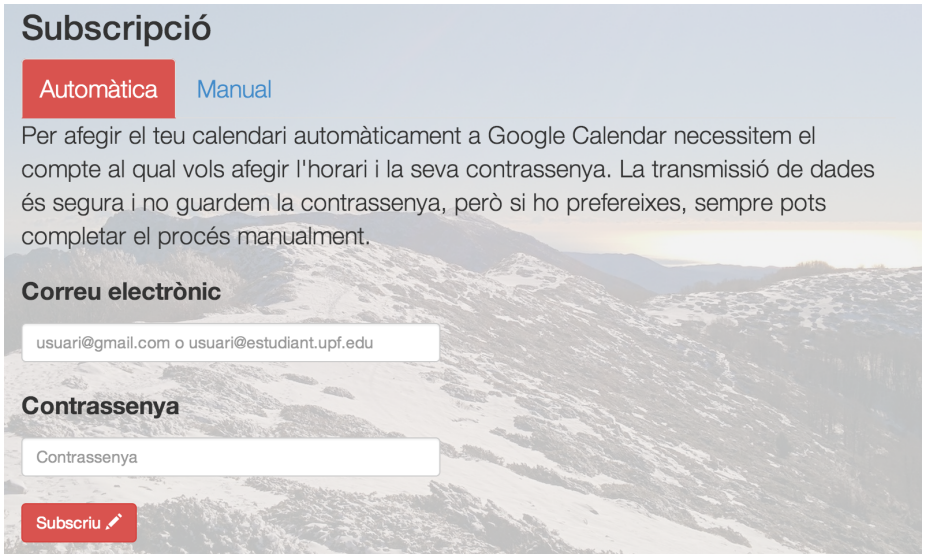


Figure 5.2: *Horaris Pompeu* automatic subscription form.

Automation

I added a new script, called `subjectstojson.py` that performed the same tasks as the previous *subjectparser* but saved the results onto a JSON file instead of directly to the database. This way, I could manually curate the lessons list since, after all, this was not an activity which is performed often.

With that change done, *subjectparser* simply loaded the JSON file and saved the subjects onto the database.

I also moved the legacy timetable sources from a *Python* dictionary to a JSON file. The objective, in both cases, was to make the architecture of the app a

little bit more generalistic, so that using it by other faculties would only be a matter of adding the corresponding configuration files with the data sources.

Deployment

So far I haven't talked about the deployment process of a *Django* app. Since in this iteration I added some files on the repository regarding the deployment configuration, I will explain how the production system works.

Python web applications share a common protocol for communicating with web servers called WSGI[7]. Any server that implements WSGI can talk with a *Python* WSGI service. This is very useful for *Python* core web developers and HTTP server developers since, as long as everyone follows the standard, both parts can talk to each other.

The WSGI-compatible web servers can be typical web servers, such as Apache (via the WSGI module) or specialized WSGI servers. Those specialized servers are simple programs that focus on handling WSGI requests fast and little else. A typical WSGI server has a pool of workers (running the *Django* project in this case) and an event loop. In this case, dealing with requests is a matter of dispatching requests to workers and little else.

Using simple servers like this is good for performance, but it's not enough for all the project needs. Those servers can't serve static files, don't handle rewrites or can't be configured with SSL. To fill those needs, another kind of webserver will have to be used.

This project follows the typical *Django* deployment architecture. The front-facing server is Nginx, which is used as a reverse proxy to GUnicorn (WSGI server) and a static file server (for calendars). GUnicorn is used to serve the django web application.

The Nginx config handled static files serving and reverse proxying. No caching strategy was specified. SSL support is not added in this example configuration file.

[7]The formal definition of the WSGI protocol can be found on [Eby 2003] and [Eby 2010].

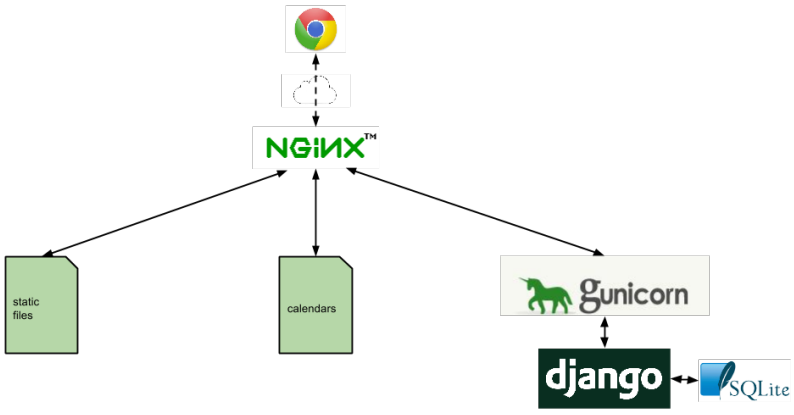


Figure 5.3: Deployment diagram of *Horaris Pompeu*

Chapter 6

Sprint 3 - Automation, user communication and analytics

This will cover the *third* sprint of the project, from *13-11-2013* until *27-11-2013*. Tagged as v3[1] on the*Horaris Pompeu*GitHub repository. Diff[2] between v2 and v3.

v3 centered around refactoring the management commands, streamlining the database schema, adding some extra views (FAQ and contact form), Google Analytics support, a few extra tweaks on the user interface and some improvements on the production configuration files.

Presentation

The changes in the presentation layer were small, in comparison to the last iteration, but a nice step towards a more refined user experience nontheless.

I added Google Analytics support, so I could track user behaviour on the page. I also changed the behaviour of the navigation bar so it would collapse on mobile devices. Cookies were used to store the options chosen on every part of the calendar creation process, and used to navigate backwards via the top navigation bar.

The new navigation bar had links to the previous steps in the process and, thanks to the cookies, it could restore the views based on the previous selected choices.

[1] Https://github.com/octavifs/horarispompeu/tree/v3.
[2] Https://github.com/octavifs/horarispompeu/compare/v2...v3.

Figure 6.1: *Horaris Pompeu* navigation bar.

Also, *Horaris Pompeu* was modified to only display subjects of the current term, so the navigation bar indicated which term was available at the moment.

Another important change as far as presentation is concerned was the addition of a FAQ page and a contact page. The FAQ page was simply an expanded explanation on what *Horaris Pompeu* was and how it worked. The contact form was a simple way to retrieve feedback from users. Those new pages needed their own templates, view functions and the configuration of the email system (the contact form contents are sent to an email account I created for *Horaris Pompeu*).

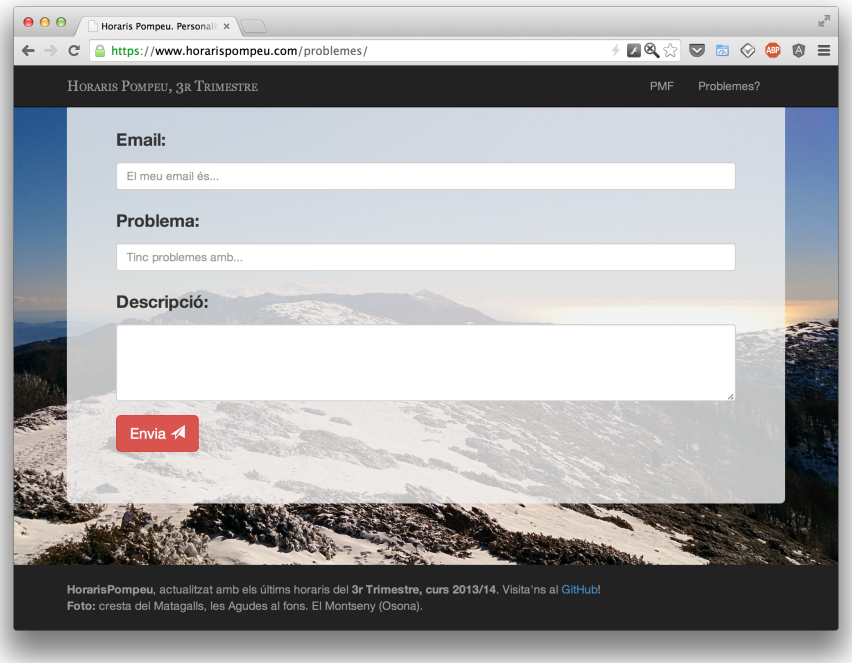


Figure 6.2: *Horaris Pompeu* contact form.

Parsing

The parser was refactored, using generators instead of lists (better performance and conceptually cleaner). Also, the new version uses more accurate regular expressions, that catch edge cases better.

Automation

Most changes in this update are related to the management commands and the models. In the `models.py` file, apart from documenting better the purpose of some classes, I deleted **SubjectDuplicate**. As I had mentioned in the last update, the parsing of the subjects was dumped to a JSON file now, which I manually curated. Since `subjectparser` inserted the lessons in this list, if you manually make sure that the list is correct, there is no need for more checks. I remind that **SubjectDuplicate** stored entries that had been deleted in **Subject** so that re-running `subjectparser` would not add them again.

`aliasparser` and `initdb` were deleted, while `lessonparser` inherited their functionality. The new `lessonparser` simply stopped if it encountered a subject without an alias when parsing lessons. The function in charge of updating calendars was moved onto another command (`calendarupdater`) and most critical operations (lesson inserts and deletes) were moved to a new `operations.py` file, which made the functions easier to test.

Merging

Since synchronizing the legacy timetables with the local database is conflictive and error prone, I started a test suite to ensure that the implementation worked fine against specific known cases. In this case, when the implementation has to deal with so many edge cases, it is very interesting to have a series of automated tests to prove that the implementation works and, even more importantly, that future refactoring of the library won't break old expected behaviours.

To perform good unit tests, having small and independent pieces of functionality is key (so tests are kept small, to the point, and are difficult to get wrong). The problem was that, since all the insertion and deletion functionality were methods in a Django Command class, using them as standalone functions for the unit tests was not ideal. This is the reason why I moved that code into a new `operations.py` file, composed only by functions that needed no previous context.

The new `operations.py` releases `aliasparser` and `lessonparser` of the following tasks:

- insert lessons
- delete lessons
- insert subject alias
- insert degree subject
- update calendars

The implementation of those tasks is almost the same as it was in the previous iteration, with the exception that `delete_lessons` got better at recognizing manually updated Lessons, if only the date had been modified.

The newly created unit tests, stored under the `timetable/tests` folder reproduce a series of real-world cases the code has to deal with. For details about those cases I recommend to check the unit tests themselves (operations tests[3], parser tests[4]).

Deployment

Improved the default nginx configuration file, with request compression by default and static file caching configured. I also added gunicorn (WSGI server) to the django project itself, so it could be called using the `./manage.py` command.

[3] Https://github.com/octavifs/horarispompeu/tree/v3/timetable/tests.
[4] Https://github.com/octavifs/horarispompeu/blob/v3/timetable/tests/test_parser.py.

Chapter 7

Sprint 4 - Simpler parsing strategy, better parsing accuracy

This will cover the *fourth* sprint of the project, from *27-11-2013* until *08-02-2014*. Tagged as v4[1] on the *Horaris Pompeu* GitHub repository. Diff[2] between v3 and v4.

v4 was the first widely used version of *Horaris Pompeu* and the first one I could start analyzing user behaviour thanks to the addition of Google Analytics and the server logs.

v4 includes a refactored script for automatic calendar subscription, Nginx configuration with SSL support, automatic backup strategy, better documentation (regarding the subscription process), new (more reliable) parsing strategy, streamlined database schema and random background images.

Presentation

The most relevant change, regarding presentation, was the refactor of the PhantomJS script (the one in charge to automatically subscribe users to calendars). The old script used click events to perform the user actions (as if the user itself did the clicks), which happened to be quite inconvenient, since it caused some conflicts with Google Calendar javascript framework, and to perform a click on the user interface you had to follow a special sequence of events (hover -> press

[1] Https://github.com/octavifs/horarispompeu/tree/v4.
[2] Https://github.com/octavifs/horarispompeu/compare/v3...v4.

-> release). For extra details, check the source[3] of `/casperjs/addICSCal.js`. The new version, instead of simulating clicks, it simply performed the necessary network calls. This was simpler, more reliable and faster, since the original option needed some timeout between events so that the interface could be drawn. I also could avoid using CasperJS, a library on top of PhantomJS to simplify user form filling and other actions and worked with a bare copy with PhantomJS plus an injected jQuery script to perform DOM manipulation and AJAX calls. For more details, check the source[4] code of the script at `/phantomjs/gcal_addICS.js`.

Apart from the new automatic subscription script, I also performed some cosmetic changes. The new version of the project was able to load backgrounds randomly from a list of images defined by the project settings. Each image persists during the length of the user session in the site (using cookies) and has a small footer with the description and location. I also made sure I had the copyright of all the images shown, to avoid any sort of licensing infringement. Images were resized and optimized so the filesize would not be above 512KB and the site would keep loading fast.

Finally I added screencasts to display the process of adding a calendar manually from within the Google Calendar interface and another one displaying how to synchronize the calendar with an Android device.

Parsing

Parsing legacy timetables right was a problem I had been trying to solve since the first iteration but could never get completely right. The error rate of the parser was 5%, aprox, which in a typical term amounts to 140 lessons out of 2900. 140 out of 2900 may not seem that much, but having to modify them manually it's slow and difficult to complete, especially because the parser may think that an entry is right, when it's not, so you can't detect it and correct the error in advance.

Since I was not yet satisfied with the success result of the parser and the automatic handling of conflicts I branched the repository and started a new, simpler, implementation of the parser.

The major problem with the old parser was correctly detecting the groups and classrooms. There are many different formats used accross all subjects and calendars and new ones are always added. Dates, on the other hand, while sometimes tricky to parse, it's easier to get them right. With that in mind, I started to balance how much of a loss it would suppose if data such as the lesson subgroup, lesson kind or room was added into a single field instead of parsed into separated ones. The main problems if information is aggregated is that you can't show lessons by subgroup only and that the classroom can't be put into the

[3] Https://github.com/octavifs/horarispompeu/blob/v3/casperjs/addICSCal.js.
[4] Https://github.com/octavifs/horarispompeu/blob/v4/phantomjs/gcal_addICS.js.

iCalendar location field of an event, which makes looking for that information in a calendar application more incovenient.

Not being able to filter classes by subgroup could be a problem to some, but in general is a fair trade-off. Having a subgroup chooser wouldn't be great from a usability standpoint (too many of them, usually more than one per lesson kind) and a lot of times would not even be that useful for the students themselves, whom usually attent to other practice or seminar groups than their own. With that in mind, not parsing subgroups seemed like an acceptable sacrifice in functionality if it helped bringing up the parser's precision.

The other problem would be that not parsing lesson classrooms makes it impossible to put them into the location field of an iCalendar file. Usually calendar applications show the location of an event even when its description is too long, which is convenient for a lot of lessons that have very long subject names (and there's a great deal of them). If, instead of putting some information on the location field of an event, all the information is placed into the description, it is cut on the weekly or monthly view of the calendar. That is a problem, from a usability standpoint, since the users have to click on the event to expand it and be able to see the correct classroom and group (1 extra action by their part). Still, 1 extra click by the user may be a fair trade-off if the information he gets is 100% accurate.

If we take another look onto a typical lesson, as written in a legacy timetable, we always find something like this

```
Subject name
general info (usually lesson kind)
date
specific info (usually groups and classrooms)
more dates
more specific info
```

The parser acknowledges this and creates a lesson for each pair of date and specific info it finds. This way, the lesson object drops the fields *kind*, *group* and *room* and adds a new, more generic, *data* field.

Thanks to this new parsing strategy, even though the data displayed in the students calendars is not as easy to read (since everything is put in the event description, and entries tend to be long), the accuracy of the new system is almost 100%. In fact, in this 3rd term, only 1 lesson out of 2910 was parsed wrong (because of a wrongly formated date that the regular expression could not catch). This big improvements in accuracy of the parser will also have big benefits regarding our merging problem and will virtually eliminate any kind of manual administration, if the update process are well automatized.

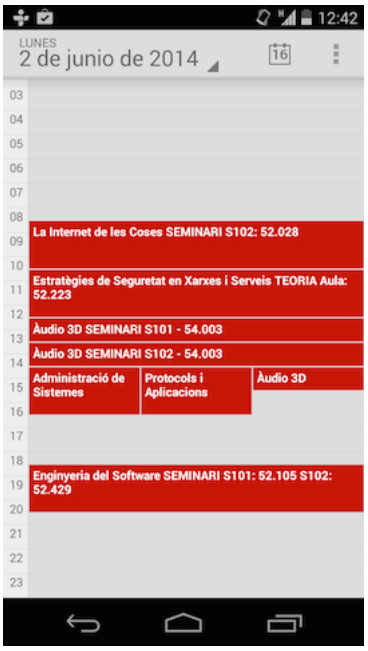


Figure 7.1: Weekly view of a calendar created through *Horaris Pompeu*. Some information is cut if no space is available.

Figure 7.2: Detail view of a calendar created through *Horaris Pompeu*.

Merging

In the previous section we have already discussed the benefits and drawbacks of the new parser regarding accuracy. In this one, we will talk about the effects of the new parsing model on the database schema and the synchronization with the legacy timetables.

The main problem merging had was that I expected to modify manually many of the entries generated automatically with the parser. This makes deletion hard since, to delete, I query the Lesson by reparsing the old HTML file, and if the Lesson has manually changed in the database it won't match with the original content. Since the new parsing virtually eliminated the need to do any kind of manual update, this problem was less likely to happen. Still, I found a way to eliminate it entirely thanks to, once again, hashes. Each Lesson class by the parser is hasheable so I added a *uuid* field to the Lesson model and saved the hash of the parsed Lesson there. This means that, even if the model Lesson is modified, if the hash matches, it still refers to the Lesson created by the parser. Deletions, with the new *uuid* field, are just a mater of deleting in the database all Lesson models whose *uuid* matches the hash of the lessons to be deleted.

Thanks to the use of hashing and the new parser, the chances of having to do manual updates or getting an update wrong are almost non existent, so all the models used to prevent information loss can be deleted. This means that *LessonArchive*, *LessonInsert*, *LessonDelete* and all the signaling system can be deleted. Not only does this change make our schema easier to understand, but it also increases its performance, since database signals aren't used.

`operations.py` was updated to ensure that insertions and deletions of lessons followed the new model described above.

Automation

A new command was added, called `updater`. This was the complete batch operation necessary to synchronize the database and calendars with the legacy timetables. It also implemented an automatic backup system. `updater` was designed as a maintenance task to be launched daily that would take care of the administration of the site and send an email to the admin informing of the process. The script does the following:

1. Backup of database and old timetables in a local folder, timestamped.
2. Backup of database and production configuration to Amazon S3 (in case the production server is lost), timestamped.
3. Parse legacy timetables in search for new aliases (using `subjectparser`).
4. Parse lessons (using `lessonparser`).
5. Update calendars (using `calendarupdater`)

6. Send email to admin detailing the results of the process

With this running daily, *HorarisPompeu* is a self maintained project, with minimal interaction by my part. Only on the start of the term or in the case of a new subject alias I need to take further action.

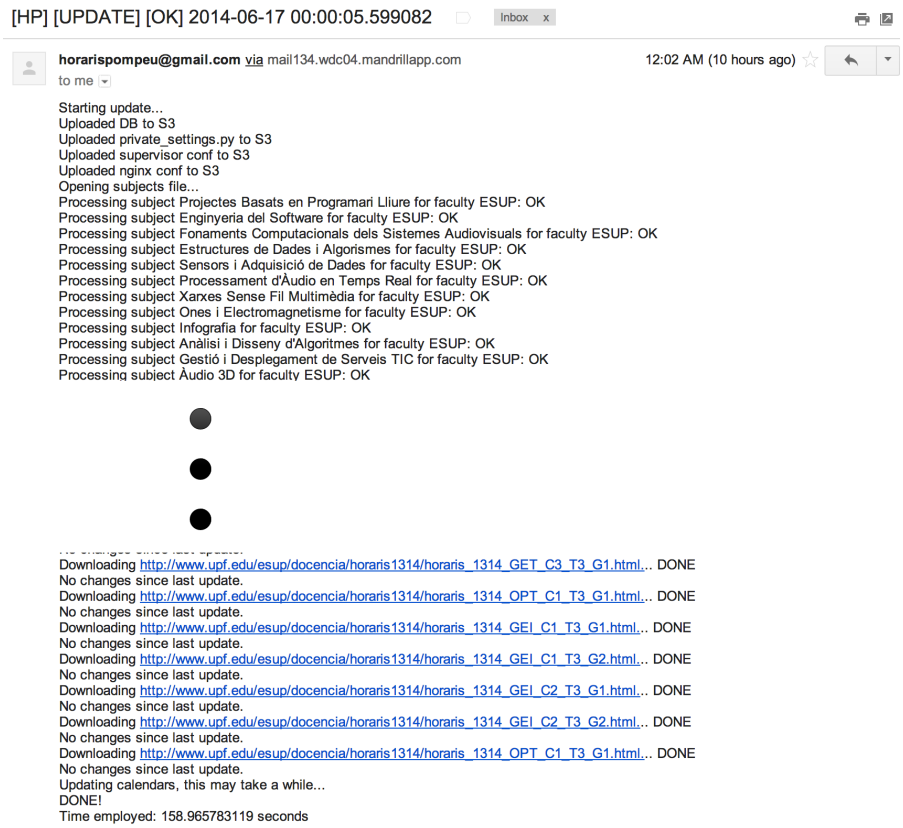


Figure 7.3: Email sent by *Horaris Pompeu* to the administrator on each `updater` run.

Analytics

Since I had added Google Analytics to keep track of the webpage usage by the users and *Horaris Pompeu* started gaining some traction the second term of this year, I will analyze some metrics regarding the real usage of the platform.

A lone Facebook post on the ESUP group on 8th February skyrocketed *Horaris Pompeu* popularity and has maintained it afloat for the rest of the year. That

sole day a 103 calendars were created. The server handled the user load very well and no problems were noticed during that usage spike.

Since then, *Horaris Pompeu* has been receiving a lower but steady flow of users, especially near the announcement date and the start of the 3rd term, which is when calendars need to be created. So far, the traffic statistics behave as expected.

Horaris Pompeu bounce rate is 31.83%. That means that almost 7 out of 10 students that visit the page take at least some action. Even more important is the fact that 5 out of 10 users that visit the page goes through the process of creating a new calendar. If we only focus on the calendar creation rate of new users (the ones visiting *Horaris Pompeu* for the first time), the rate rises to 60.58%.

Horaris Pompeu has an average time per session of 2min 48s. This could be considered the time a standard user takes to create and subscribe to his calendar. Less than 3 minutes. With an investment of less than 3 minutes in *Horaris Pompeu* any user can't forget about his schedule for the rest of the term. It is also relevant the average number of pages viewed per session, which is very close to 5 (5 is the number of steps required to create a calendar). This correlates perfectly with the high number of calendars created per session.

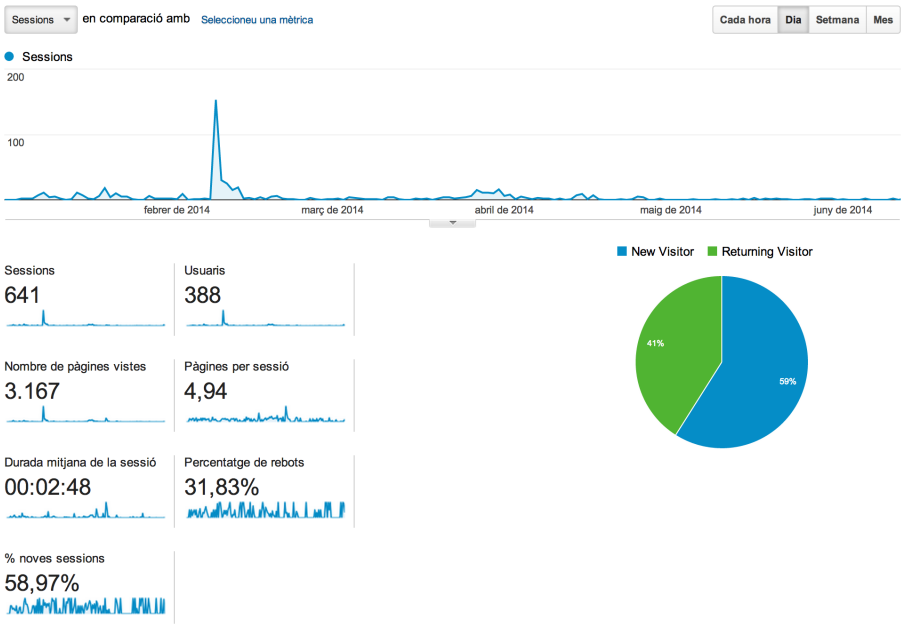


Figure 7.4: *Horaris Pompeu* metrics from *01-01-2014* until *15-06-2014*

To end this analysis, I will check the Nginx logs, in search for active calendars (calendars subscribed by clients). From the user agent we can ascertain that the

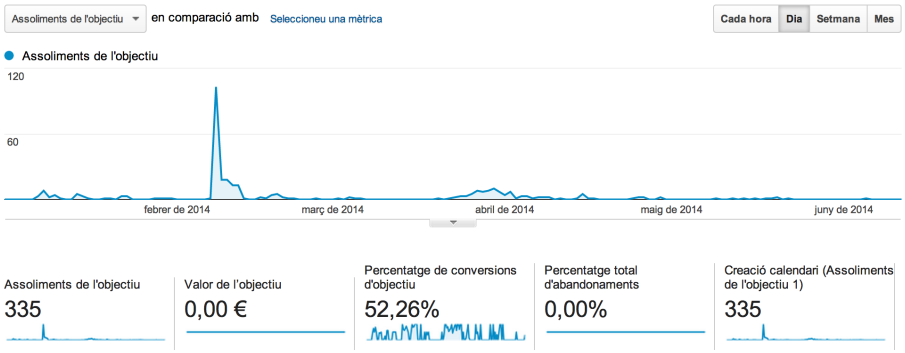


Figure 7.5: Calendars created form *01-01-2014* until *15-06-2014*

vast majority of them are used from Google Calendar, even though some of our users opt for desktop clients such as Outlook on Windows and iCal on OSX.

Using the logs form 7 different days, we can see that the number of active unique calendars is 160, and the average number of updates per day is 3.2 times per day, that is, one update every 7.5 hours[5]. 160 calendars in use out of 335 created calendars is not such a great number (48%), especially in comparison to the ones presented before. That makes me think that the decision not to use Google Calendar API may have been suboptimal and that, at least, I should rethink the communication strategy to shorten the gap between calendar creation and subscription.

Still, looking at the facts, *Horaris Pompeu* has a user conversion rate of 25%, engages almost 70% of it's users and allows the creation of calendars in 5 steps and less than 3 minutes. I think that those numbers reflect that there was demand for better schedules.

[5]This data is obtained from the Nginx logs from the last 7 days. Only unique calendar urls were counted.

Chapter 8

Sprint 5 - *Gestió Acadèmica* refactoring

This will cover the *fifth* and final sprint of the project, from *17-04-2014* until *27-04-2014*. Tagged as v5[1] on the*Horaris Pompeu*GitHub repository. Diff[2] between v4 and v5.

Next year, *Gestió Acadèmica* will give public access to their platform so that students can create their own timetables. *Gestió Acadèmica* already had that information in their database, since it was inputted to create the weekly timetables posted on each classroom of the university, but that information was not available to the public, so every faculty had implemented their own system. Starting this year though, things are changing and many faculties will adopt the new system for the timetables.

The new system, while universal for the whole university, does not allow creating timetables with multiple degrees or years. It also doesn't offer any way to subscribe to the custom calendar or any permanent url to access it so, in some ways, it's an inferior solution, with the upside that the data is nicely formatted, so it will be easier to parse this time.

Since the new system still does not solve the problems *Horaris Pompeu* was set to resolve, I decided to rewrite the parser so it could make use of the new backend, this time with the addition of new faculties and degrees.

v5 is the first iteration of *Horaris Pompeu* that uses the new platform[3] by *Gestió Acadèmica* as the data source, instead of the legacy *ESUP* timetables. This implies changes in the parser, modifications in the database schema and new management commands. In this iteration I also refactored the views, refined the

[1] Https://github.com/octavifs/horarispompeu/tree/v5.
[2] Https://github.com/octavifs/horarispompeu/compare/v4...v5.
[3] Http://gestioacademica.upf.edu/pds/consultaPublica/look[conpub]ActualizarCombosPubHora.

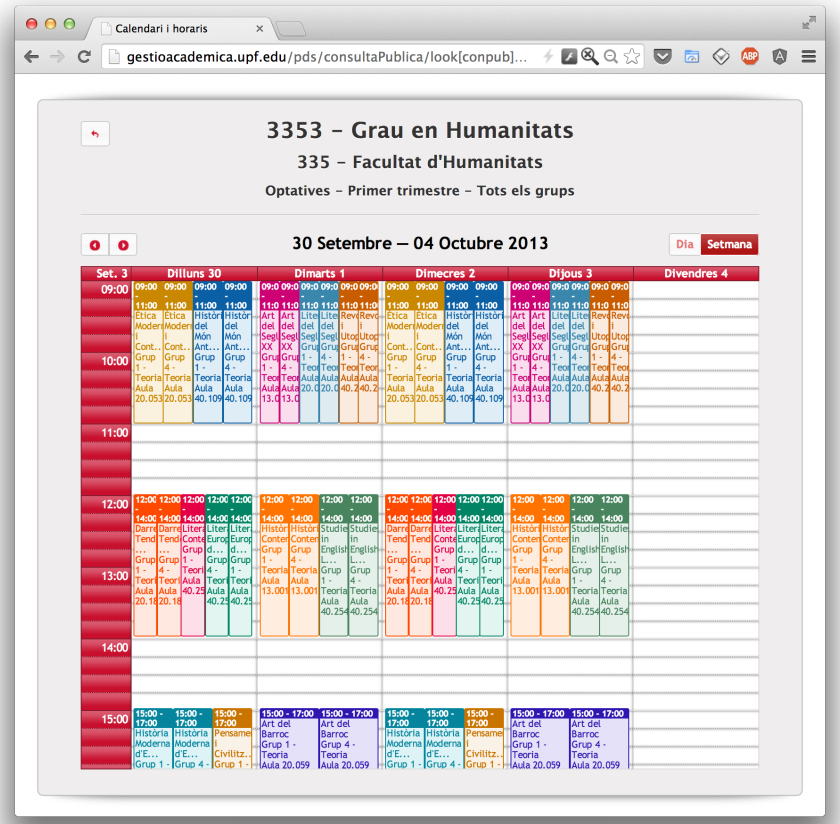


Figure 8.1: *Gestió Acadèmica* new custom calendar system.

implementation of the random background images, tweaked the CSS to reduce font size and improved the settings by clearly dividing which options should be public and which ones private.

Presentation

Regarding the presentation, the most visible change is the addition of a new faculty template. Since the new system has support for multiple UPF faculties, the first step now is choosing the faculty, instead of the degree. I also reduced the fontsize of the form elements, so that long names would not overflow as frequently.

The refactoring performed under the hood was more profound, since I wanted to implement best practices in Django developments as described in [Greenfield and Roy 2014]. First and foremost, I changed the implementation of the random background image to use a context processor instead of a view decorator. With the context processor I made sure that each request has the background variables injected in the context. This way, it's not necessary to manually add the variables with a decorator.

The implementation of the views was refactored, using class based views instead of function based views, but the functionality remained the same.

Parsing

The most important new piece of code of this v5 iteration is the scraper. `scraper.py` has a double role:

- discover all available faculties, degrees and subjects.
- get all available lessons.

To perform the first task, the scraper has a `populate_db` function. `populate_db` starts scanning the *Gestió Acadèmica* subject chooser (url) to discover all possible faculties, degrees and subjects. To do this, the scraper scans all possible options applying a Depth-First-Search strategy from the root of the tree.

```
populate_db tree:
+- planDocente (1:N)
`+- centro (1:N)
 `+- estudio (1:1)
  `+- planEstudio (1:N)
   `+- curso (1:3)
    `+- trimestre (1:2)
```

```
`+- grupo (1:N)
 `+- asignaturas
```

Once the whole tree is available, it is flattened and inserted into the database.

The other role of the scraper is to parse lessons and store them into the databse. To do this I had to reverse engineer the system used by *Gestió acadèmica* to display the data which, in this case, is quite simple.

To create a calendar a POST request to `http://gestioacademica.upf.edu/pds/consultaPublica/look` needs to be made. The parameters needed by the post request are:

```
planDocente, centro, estudio, planEstudio, curso, trimestre, grupo,
asignaturas, asignatura20052, asignatura20057, asignatura20045,
asignatura20062, asignatura20134, asignatura20137, asignatura20080,
asignatura20070, asignatura20077, asignatura20107, asignatura22037,
asignatura20114, asignatura20110, asignatura20127, asignatura22038,
asignatura20166, asignatura20098
```

It is easy to see that each new subject needs its own entry as a POST request key.

Also important to note that to query that URL, some cookies need to be set, so before performing the POST request, you should GET `http://gestioacademica.upf.edu/pds/consultaPublica/look%5bconpub%5dInicioPubHora?entra`

After the POST request is done, you will obtain an HTML with some content. Inside that content, there is a script with a big list of JSON objects, Those JSON objects are the lesson entries we need. They look like this:

```
{
    title: "Darreres Tendències Artístiques",
    aula: "20.183",
    tipologia:"Teoria",
    grup:"1",
    start: "2013-09-25 12:00:00",
    end: "2013-09-25 14:00:00",
    className:"assig03 ",
    festivoNoLectivo: false,
    publicarObservacion:"N",
    observacion:""
}
```

It is important not to interpret the page, since when the javascript executes the array is deleted and the data can only be accessed through the browser's DOM.

To perform all this actions with *Python*, it is a simple matter of using the requests library, with care to ensure that cookie sessions are maintained. The parsing of

the JSON array is done via regular expressions, since it was the easiest way to isolate that data from all the boilerplate code from the request.

Now that we know how the *Gestió Acadèmica* service works, implementing `populate_lessons` is just a matter of performing the correct requests to retrieve the desired lessons. I also made sure that the implementation could fetch multiple URL's at once, to speed up the retrieval process of the data.

Merging

Each update drops the Lessons database and loads it from scratch. Since we assume that *Gestió Acadèmica*'s data is correct and that we haven't manually modified anything we might want to keep, the simplest strategy is to replace our database contents by what the scraper has retrieved.

Automation

Data in the new system is well defined, so there is no need to deal with subject aliases or other typos that might arise from human errors. This means that the SubjectAlias table is refactored out of the system. It also means that the `subjectparser` and `subjectstojson` commands are deleted, since they are no longer necessary. The sources needed to locate the legacy timetables are also deleted, along with the `operations.py`, which dealt with insertions and deletions of the old Lesson system.

New management commands are the following:

populate.py : Bootstraps the database with faculties, degrees, subjects and lessons.

update.py : Updates lessons and calendars.

backup.py : Backs up database and config, calls `updater.py`. Reports results via email to site admin.

Conclusions

In this report we have gone through the process of designing and evolving a product to create custom calendars for UPF students. As a result, we've obtained a web application that parses the legacy timetables accurately, manages itself and enables custom calendar creation for a whole term in less than 3 minutes.

Development cycle has been defined by short bursts (usually one or two weeks, rarely longer than a month), followed by long periods of inactivity where I could focus on managing the platform and evaluating the usefulness of the changes I had made. The large number of iterations, the time spent managing the system and the feedback collected by the users has proven very useful to pin-point the best approach to the problem.

Attention to detail is time consuming. Even though the foundations of the user interface were established as far as *v1*, all iterations have included some changes in the templates, whether they were more superfluous, such as dynamic background images or relevant, like automatic calendar subscription.

Automation is key. Having an automated system in charge of updating the calendars and backing up the service daily frees me from worries and *Horaris Pompeu* from errors. Every minute dedicated to it will return its investment tenfold.

Sacrifice is a powerful tool, when judiciously applied. Thanks to a relaxation on the parser requirements (*v4*) I was able to simplify the codebase, increase the accuracy of the parser and, in conclusion, end up with a much more reliable and automatable system, while losing little functionality.

Launching a critical service challenging. Even though I advertise *Horaris Pompeu* as an unnoficial solution, if it ever has any hope of gaining traction it has to be 100% reliable. This is the main reason why I delayed public announcement as far as *v4*, because that was the first version were I was truly convinced that all lessons were right. I have received no complaints so far of any missed classes because of *Horaris Pompeu*, and I'm sure I would have gotten notice, since errors and problems are the main (and only) motivation for users to talk with the support team (me). Still, the feedback has been positive and having users, while a responsibility, it is also a big motivation to continue improving and

maintaining the system. The sooner you can enter this feedback loop the better. Your software will thank you for it.

Now that the analytics have validated *Horaris Pompeu* approach, I plan to do a university wide launch next year. Since the coding part is finished it will only be a matter of setting up the web app and focus on doing a good marketing campaign. If anything, the system leaves us with a sole open question: how do we engage more users to finish the subscription process to their calendars? Right now about 50% percent of the students creating a calendar subscribes to it, which may be either because they don't understand the system or because they don't trust the automatic subscription approach. It would be interesting, for upcoming terms, to evaluate how a better communication strategy may raise the subscription rates or even to reassess the usage of the Google Calendar API as a means of lowering the barrier to subscription.

Bibliography

B. Desruisseaux, E. and Oracle. 2009. Internet Calendaring and Scheduling Core Object Specification (iCalendar). http://tools.ietf.org/html/rfc5545.

Beck, K., Beedle, M., van Bennekum, A., et al. 2001. Agile Manifesto. http://agilemanifesto.org/.

C. Daboo, E. and Apple Inc. 2009. iCalendar Transport-Independent Interoperability Protocol (iTIP). http://tools.ietf.org/html/rfc5546.

C. Daboo, E. and Apple Inc. 2013. Parameter Value Encoding in iCalendar and vCard. http://tools.ietf.org/html/rfc6868.

Eby, P.J. 2003. Python Web Server Gateway Interface v1.0. http://legacy.python.org/dev/peps/pep-0333/.

Eby, P.J. 2010. Python Web Server Gateway Interface v1.0.1. http://legacy.python.org/dev/peps/pep-3333/.

Foundation, T.A.S. 2004. Apache License, Version 2.0. http://www.apache.org/licenses/LICENSE-2.0.html.

Free Software Foundation. 1991. GNU General Public License version 2. http://www.gnu.org/licenses/gpl-2.0.txt.

Free Software Foundation. 2007. GNU General Public License v3. http://www.gnu.org/licenses/gpl-3.0.txt.

Github. 2014. Open source licensing. https://help.github.com/articles/open-source-licensing.

Greenfield, D. and Roy, A. 2014. *Two Scoops of Django: Best Practices For Django 1.6*. Two Scoops Press.

Information, F. and Standards, P. FIPS PUB 180-4 Secure Hash Standard (SHS). March 2012.

Krasner, G.E. and Pope, S.T. 1988. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. .

Kruchten, P. 2003. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional.

Leff, A. and Rayfield, J.T. 2001. Web-application development using the model/view/controller design pattern. *... , 2001. EDOC'01. Proceedings. Fifth IEEE ...*, 118–127.

Open Source Initiative. 1988. MIT License. http://opensource.org/licenses/MIT.

Rubin, K.S. 2012. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional.

Sims, C. and Johnson, H.L. 2012. *Scrum: a Breathtakingly Brief and Agile Introduction*. Dymaxicon.

Sommerville, I. 2010. *Software Engineering*. Addison-Wesley.