University of Ljubljana
Faculty of Computer and Information Science

Miran Levar

Web-based platform for dataflow processing

master's thesis

second-cycle study programme
computer and information science

Supervisor: prof. dr. Blaž Zupan
Co-supervisor: Denis Helic, Assoc.Prof. Dipl.-Ing. Dr.techn.

Ljubljana, 2015

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Miran Levar

Spletno okolje za podatkovno vodeno procesiranje

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Blaž Zupan
Somentor: Denis Helic, Assoc.Prof. Dipl.-Ing. Dr.techn.

Ljubljana, 2015

Declaration of Master's Thesis authorship

I, the undersigned Miran Levar, am the author of this Master's Thesis entitled:

Web-based platform for
dataflow processing

With my signature, I declare that:

- the submitted Thesis is my own unaided work under the supervision of prof. dr. Blaž Zupan and co-supervision of Denis Helic, Assoc.Prof. Dipl.-Ing. Dr.techn.,
- all electronic forms of this Master's Thesis, the title (Slovenian, English), abstract (Slovenian, English), and the keywords (Slovenian, English) are identical to the printed form of the Master's Thesis,
- I agree with the publication of the electronic form of this Master's Thesis in the collection "Dela FRI".

In Ljubljana, 26$^{\text{th}}$ September 2015 Author's signature:

Acknowledgements

Thanks to Prof. Zupan for his support and guidance through the years as well as the opportunities he has helped create for me. Thanks to Prof. Helic for overseeing my year in Graz and collaborating with me on this work. Thanks to the Genialis team for inviting me to participate in an incredible project. Finally, thanks to everyone else, friends and family, who supported me during my studies.

Miran Levar

Kazalo

List of abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BSON	Binary JSON
CSS	Cascading Style Sheets
DBMS	Database Management System
DOM	Document Object Model
FBP	Flow-Based Programming
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IT	Information Technology
JSON	JavaScript Object Notation
MVC	Model, View, Controller
NoSQL	Not only SQL
ORM	Object-Relational Mapping
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
URL	Uniform Resource Locator
VPL	Visual Programming Language
WS	Web Service
XML	Extensible Markup Language
YAML	Yet Another Markup Language, YAML Ain't Markup Language

Abstract

This Thesis presents the design and implementation of a web-based general-purpose data processing platform. The platform consists of a storage system, processing system, and an application server with an exposed RESTful interface. Through this interface clients, such as web browser applications, can access and control data processing as well as view the results of it. The processing system is inspired by dataflow processing, defining each process as a *black box* with inputs and outputs. Connecting the outputs to inputs of other processes enables data to flow between processes and allows for creation of processing pipelines. Triggers – conditions under which a process can begin work on available input data – enable automation of the processing pipelines. Such a system allows users who lack programming expertise to easily create and run complex dataflow operations.

Keywords

web application, multi-layered architecture, dataflow processing, data analytics

Povzetek

V magistrski nalogi je predstavljena zasnova in implementacija pilotne verzije splošne spletne platforme za podatkovno analitiko. Podrobno sta predstavljena strežniški del za hranjenje in obdelavo podatkov ter programski vmesnik, preko katerega spletni odjemalec komunicira s strežnikom. S preprostimi primeri prikažemo, da je platforma razširljiva – enostavno je mogoče razširiti obstoječe ter dodati nove zmogljivosti. Sistem procesiranja podatkov temelji na podatkovno vodenih (angl. *dataflow*) principih in s pomočjo sprožilcev, ki določajo pogoje pri katerih se lahko proces samodejno prične, omogoča avtomatizacijo procesiranja. Vsak proces lahko razumemo kot črno škatlo z definiranimi vhodi in izhodi, procese pa lahko med seboj smiselno povežemo in omogočimo, da podatki iz izhodov *tečejo* v vhode naslednjih povezanih procesov. Sistem uporabnikom omogoča, da tudi brez programerskega znanja sestavijo in izvajajo kompleksne podatkovno vodene operacije, saj je abstraktne predstavitve procesov moč med seboj povezati tudi brez poznavanja njihovih podrobnosti.

Ključne besede

spletna aplikacija, več-slojna arhitektura, podatkovno vodeno procesiranje, podatkovna analitika

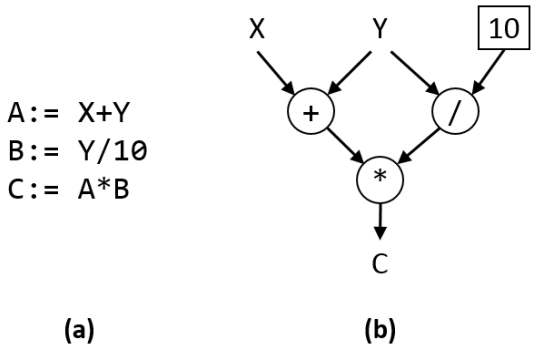
Razširjeni Povzetek

Magistrska naloga predstavi razvoj spletne platforme za hranjenje, obdelavo in vizualizacijo podatkov. Z zasnovo rešitve smo želeli izdelati sistem, ki je lahko nadgradljiv in preprost za uporabo, sploh za uporabnike, ki nimajo programerskih izkušenj. Avtor naloge je pri načrtovanju in razvoju platforme sodeloval s podjetjem Genialis[1] ter na podlagi diskusije s sodelavci razvil ogrodje in prvo iteracijo spletne aplikacije s pripadajočim spletnim strežnikom, nato pa še leto sodeloval pri izpopolnjevanju in nadgrajevanju rešitve.

Osnova delovanja platforme je podatkovno vodeno procesiranje (angl. *dataflow processing*) [17, 32], pristop, ki aplikacijo predstavi kot usmerjen graf. Med vozlišči grafa, ki predstavljajo operacije, se po usmerjenih povezavah *pretakajo* podatki. Ko v neko vozlišče prispejo (vsi) vhodni podatki, se nad njimi izvede operacija, nato pa iz izhodov vozlišča naprej po grafu do naslednjih povezanih vozlišč *stečejo* izhodni podatki. Ta pristop k procesiranju omogoča istočasno izvajanje med seboj neodvisnih operacij in je zelo drugačen od tradicionalnega, von Neumannovega, pristopa k programiranju, kjer se operacije izvajajo ena za drugo. Slika 1 prikazuje primerjavo klasičnega programa (zaporedja ukazov) ter ekvivalentnega podatkovno vodenega programa, predstavljenega z usmerjenim grafom.

Podatkovno vodeno procesiranje je bilo na začetku preučevano na nivoju strojnih ukazov in kot alternativa von Neumannovemu modelu procesiranja. V devetdesetih so raziskovalci ugotovili, sta si pristopa med seboj

[1] `www.genialis.com`



Slika 1: Primerjava preprostega sekvenčnega programa (a) ter ekvivalentnega podatkovno vodenega programa, predstavljenega z usmerjenim grafom (b), primerjava je prikazana kot so jo predlagali Johnston in sod. [17].

nista izključujoča, temveč le dve skrajnosti razpona možnih arhitektur [24]. Podatkovno vodeno procesiranje lahko v kontekstu klasičnega procesiranja razumemo kot večjedrno arhitekturo, ki vsak strojni ukaz izvede v svoji niti. Ta nov pristop je podlaga za preučevanje tako imenovanega hibridnega podatkovnega procesiranja (angl. *hybrid dataflow*), z vidika naloge pa je pomemben, ker kaže, da je podatkovno vodeno procesiranje moč razumeti tako na nivoju strojnih ukazov kot tudi na višjih, abstraktnejših nivojih.

Programiranje na osnovi tokov (angl. *flow-based programming,* FBP), ki ga je v sedemdesetih izumil Morrison [22, 23], je nekoliko višje-nivojski pristop k podatkovno vodenemu procesiranju. FBP razdeli razvoj aplikacije na dva dela – izdelavo usmerjenega grafa oz. diagrama ter implementacijo posameznih komponent (vozlišč). Posamezne komponente lahko razumemo kot črne škatle z določenimi vhodni in izhodi, ki iz vhodnih podatkov ustvarijo izhodne na podlagi nekega algoritma. Ta algoritem je lahko poljubno preprost ali kompleksen, združevanje preprostejših komponent grafa pa pomeni ustvarjenje višje-nivojskih komponent. Visoko-nivojske komponente so osnovni gradniki, s katerimi na platformi opisujemo procesiranje podatkov.

Ker podatkovno vodeno procesiranje in programiranje uporablja grafe, je le-te moč na preprost način vizualizirati. Vizualni programski jeziki (angl.

visual programming languages) [17] uporabljajo podoben pristop in z uporabo takšnih grafov oz. diagramov dosežejo izboljšanje preglednosti in razumevanja napisane izvorne kode. Vizualni pristopi k programiranju premoščajo razhajanje med podatkovno vodeno fazo načrtovanja aplikacije in deklarativno fazo implementacije [1]. Dodatna prednost vizualnega pristopa je, da so diagrami potekov lahko razumljivi tudi nepoznavalcem. Idejo in delovanje sistema je moč razumeti tudi brez poznavanja podrobnosti implementacije posameznih komponent.

Pred predstavitvijo platforme je potrebno definirati nekaj terminov, ki se pogosto uporabljajo v kontekstu platforme. **Podatek** je skupek relevantnih informacij, ki predstavlja en vnos v podatkovni bazi. K enemu podatku lahko pripada ena ali več datotek, odvisno od definicije podatkovnega tipa posameznega podatka. Del pripadajočih informacij je nespremenljiv in določen ob dokončnem zapisu podatka v bazo, drugi del pa lahko uporabniki po želji spreminjajo. Vsak podatek nastane kot rezultat **procesorja**, koncepta, ki smo ga povzeli iz podatkovno vodenega procesiranja. Procesor lahko na eni strani sprejme vhodne podatke, na drugi pa vedno ustvari vsaj enega ali več novih izhodnih podatkov. Obenem je procesor oz. sistem za upravljanje procesiranja tisti, ki določa nespremenljiv del informacij posameznega podatka.

Še en pomemben koncept so **sheme** in **predloge**, ki služijo kot opis strukture informacij shranjenih v podatku. Tako sheme kot predloge delujejo po enakem ključu, razlika v imenih pa je prisotna za lažje razločevanje med deli strukture, ki jih uporabnik lahko (predloge) in ne more spreminjati (sheme). Z zamenjavo predloge lahko uporabniki prilagodijo del strukture informacij podatka svojim potrebam. Razlog za ločeno hranjenje strukture in informacij samih je, da je s takšnim pristopom mogoče razviti sistem za generiranje vnosnih in prikaznih obrazcev. Hkrati to zelo pripomore k splošnosti platforme, saj je neglede na tip podatka mogoče shraniti vse relevantne informacije.

Zadnji koncept so **aplikacije**. Platforma svoje zmogljivosti izpostavlja v splet preko aplikacijskega programskega vmesnika (angl. *application programming interface,* API), za krmiljenje in interakcijo pa so potrebni odje-

malci. Aplikacija je uporabniški vmesnik platforme, ki preko APIja krmili procesiranje ter prikazuje rezultate le-tega. Pričakujemo, da bodo aplikacije tipično razvite za spletne brskalnike, četudi je do spletnega vmesnika moč dostopati s katerokoli združljivo tehnologijo. Nekaj primerov aplikacij je razvil Genialis (na primer osnovno okolje za upravljanje s podatki, procesi in sprožilci ter specializirano aplikacijo za vizualizacijo genskih ekspresij), razvijejo pa jih lahko tudi dovolj vešči uporabniki.

Platforma je bila zasnovana z več-slojno arhitekturo (angl. *multi-layered architecture*), ki jo sestavljajo sloj trajnosti (angl. *persistence layer*), sloj za procesiranje podatkov (angl. *processing layer*), aplikacijski sloj (angl. *application layer*) in predstavitveni sloj (angl. *presentation layer*). Cilj je bil ustvariti modularno strukturo s šibko sklopljenimi komponentami, ki med seboj komunicirajo preko vmesnikov in jih je moč zamenjati za alternativne komponente z združljivimi vmesniki. Slika 2 prikazuje diagram arhitekture sistema s pomembnejšimi uporabljenimi tehnologijami za vsak sloj.

Slika 2: Arhitektura sistema z uporabljenimi tehnologijami.

Sloj trajnosti skrbi za hranjenje podatkov. Platforma uporablja tri načine shranjevanja podatkov: relacijsko podatkovno bazo *PostgreSQL*[2], ne-relacijsko podatkovno bazo *MongoDB*[3] in datotečni sistem. Podatki z relacijsko struk-

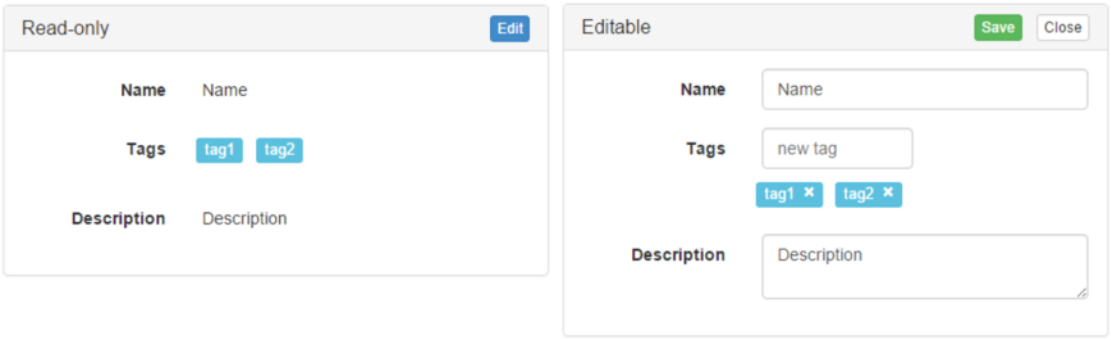
[2] `www.postgresql.org`
[3] `www.mongodb.org`

turo, kot na primer uporabniki, skupine uporabnikov in podatki o aktivnostih uporabnikov, so shranjeni v relacijsko bazo. Informacije o podatkih, ki so na platformo naloženi za procesiranje, so shranjene v MongoDB, ne-relacijski bazi specializirani za hranjenje dokumentov. Podatkom pripadajoče datoteke so shranjene v porazdeljenem datotečnem sistemu. Dostop do podatkovnih baz je krmiljen preko sistemov za objektno-relacijsko preslikavo (angl. *object-relational mapping, ORM*), kar pripomore k šibki sklopljenosti sloja trajnosti in aplikacijskega sloja.

Aplikacijski sloj je razvit v *Djangu*[4], okolju za razvoj spletnih aplikacij v programskem jeziku *Python*. Aplikacijski sloj določa funkcionalnosti platforme ter jih preko APIja izpostavlja spletu. API je definiran glede na določila specifikacije REST [11] in na zahtevke odjemalcev odgovarja s pošiljanjem relevantnih podatkov v notaciji JSON. Pomemben del zagotavljanja pravilnega delovanja sloja je avtomatsko testiranje modulov. Za večino komponent, predvsem pa API, so napisani testi, ki preverijo pravilnost delovanja teh komponent. Testi se samodejno zaženejo, ko so v izvorno kodo dodane spremembe. Sloj za procesiranje je prav tako napisan v programskem jeziku *Python* z uporabo knjižnice *Celery*[5]. Posamezni procesorji se izvajajo znotraj izoliranih Linux okolj (angl. *Linux Containers, LXC*). Tako je procesorjem onemogočen dostop do preostalega sistema ter do procesov ostalih uporabnikov.

Za avtomatizacijo procesiranja uporabljamo tako imenovane *sprožilce* [18]. Sprožilec določa pogoje, pri katerih se lahko, ko so prisotni vsi vhodni podatki, procesiranje samodejno prične. Uporabniki lahko definirajo serijo sprožilcev ter tako ob naložitvi primernih podatkov sprožijo samodejno procesiranje vse do končnih rezultatov. Avtomatsko procesiranje je posebej dobrodošlo v primerih, ko posamezni procesi trajajo precej časa, saj sistem brez nadzora in potrebe po dodatni interakciji ustvari končne rezultate. Obenem lahko neveščim uporabnikom podatkovno vodene cevovode nastavijo skrbniki

[4] `www.djangoproject.com`
[5] `www.celeryproject.org`



Slika 3: Slika prikazuje Shemo 4.2 izrisano v načinu samo za branje ter v načinu za urejanje.

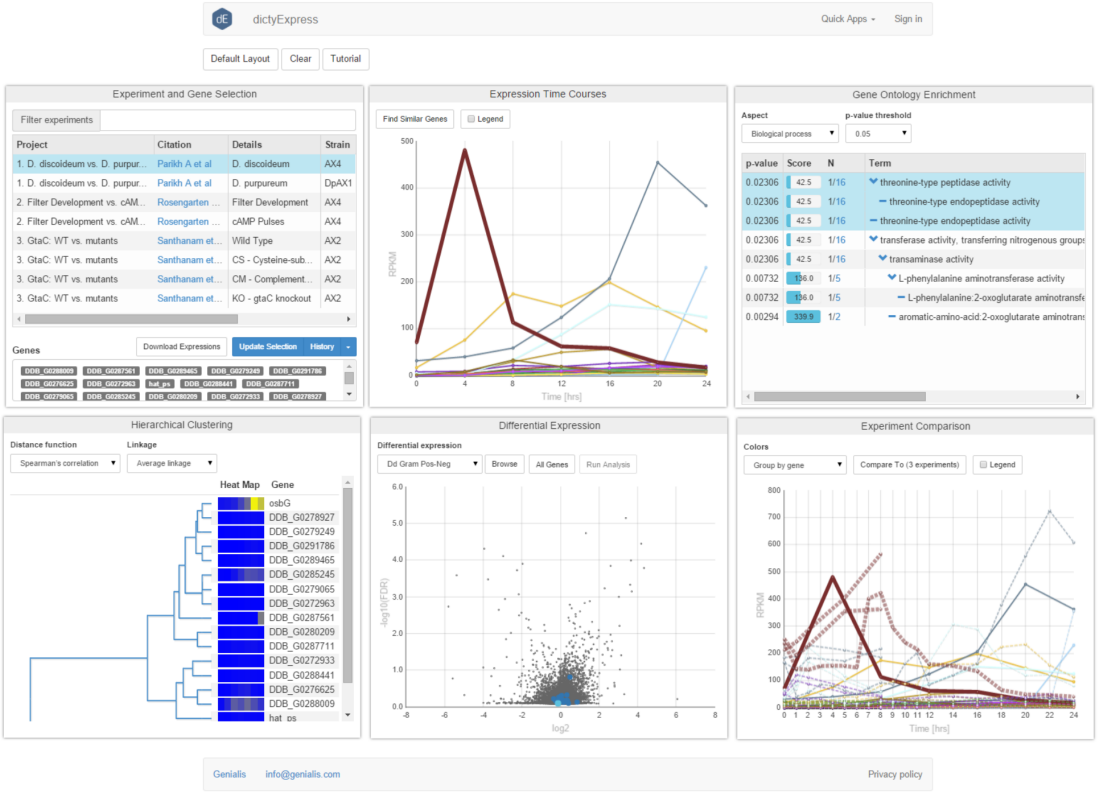
ter jim tako omogočijo nemoteno delo.

Na strani odjemalca so aplikacije predstavitvenega sloja napisane s pomočjo ogrodja *AngularJS*[6], ki smo ga izbrali kot najbolj ustreznega izmed podobnih rešitev za pisanje uporabniških vmesnikov. Ogrodje omogoča razširitev standardnega jezika HTML s tako imenovanimi *direktivami*, ki jih okolje ogrodja interpretira ter izriše glede na njihovo definicijo. S takšnimi direktivami smo na primer implementirali avtomatsko generiranje obrazcev iz shem ter informacij nekega podatka (Slika 3).

Platforma je relativno preprosto razširljiva, saj ji je moč dodati tako nove procesorje, ki so zapisani v jeziku YAML (primer definicije procesorja je Izpis 4.4), kot tudi nove aplikacije (preprost primer aplikacije prikazuje Slika 4.4). Seveda pa je mogoče izdelati tudi bistveno bolj zahtevne aplikacije – Slika 4 prikazuje zgoraj omenjeni primer vmesnika aplikacije za vizualizacijo genskih ekspresij.

V nalogi predstavljena platforma je tudi po približno dveh letih in pol od pričetka razvoja še vedno v uporabi ter se aktivno vzdržuje in nadgrajuje. Podjetje Genialis tudi razvija nove aplikacije ter tako rešuje probleme podatkovne analitike na področju bioinformatike. Predstavljene ideje so preživele in zaživele in se še vedno izpopolnjujejo v dinamičnem start-up okolju.

[6] `angularjs.org`



Slika 4: Vmesnik aplikacije za vizualizacijo in analizo genskih ekspresij *dictyExpress*[8].

[8]Aplikacija je dostopna na `dictyexpress.research.bcm.edu`.

Chapter 1

Introduction

In early 2013 a start-up company Genialis[9] set out to build a general data processing platform. The platform should support the whole data flow from uploading the raw data to viewing the results of processing in a web application. The author of this Thesis was one of the developers that planned the architecture, implemented the first iteration of the platform, and continued improving it during the first year of deployment.

The fundamental idea of the platform was to enable users that have little to no programming knowledge access to complex tools, ideally with an intuitive visual interface similar to those in data mining suites like *Orange* [8] and *KNIME* [3]. The principles behind the idea are related to (visual) dataflow programming [17, 22], which is a programming paradigm that represents every process as an abstract black box with inputs and outputs (sometimes with exposed processing parameters as well). Outputs of one black box can be fed as inputs into the next one and in this way a processing pipeline can be built that transforms the raw data into results (Figure 1.1). The black boxes, and the processes they represent, are algorithms transforming some input data into outputs and have to be developed by programmers or users with sufficient skills. Once developed, however, these black box processes can be used by anyone who can meaningfully interconnect them.

[9] `www.genialis.com`

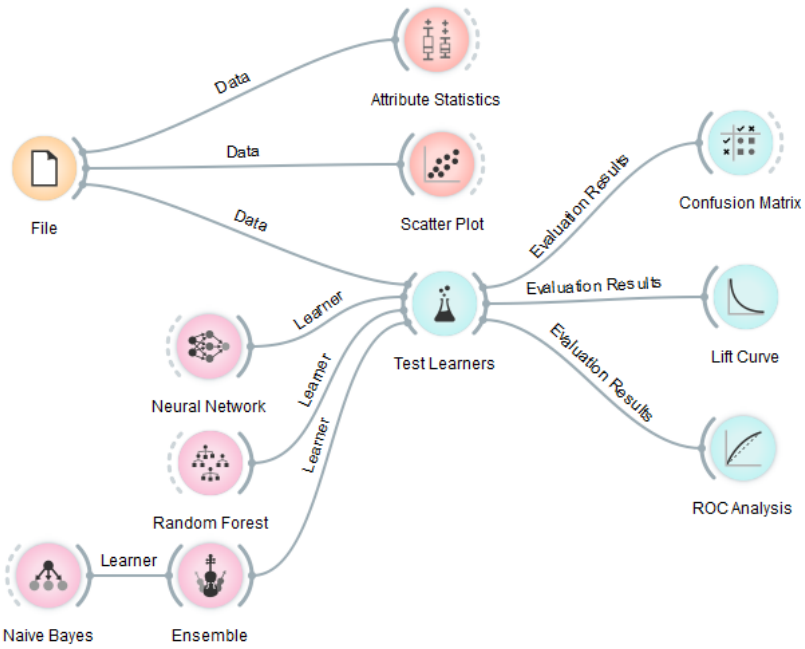


Figure 1.1: An example workflow created with *Orange*, a data analysis framework. The workflow was created with a visual programming interface. The input data File can be analysed (Attribute Statistics), visualised (Scatter Plot), and used as input data for machine learning techniques (Test Learners). The results of this learning can be further analysed and visualised (the nodes to the right of Test Learners).

Building the pipelines and making sure that all the relevant data is processed can still be tedious for the user. Because of that, automation of the process was envisioned with the use of *triggers* – conditions under which a process can be started. Users can set up triggers that automatically run processes on relevant input data. Chaining such triggers can create pipelines. This way, a user only has to set up a processing pipeline with its corresponding triggers once. Then, every time new relevant data is added to the platform it is automatically processed by setting off a sequence of relevant triggers. Automation is important because some processing tasks can take a long time to complete. Triggers make sure the processing can continue

without supervision. Another key aspect of triggers is the ability to handle changes to the data. By changing some parameters of a process inside a pipeline, the data that was previously created by that process is no longer consistent with its settings. Triggers can be configured to automatically update the rest of the pipeline after such changes are detected or to just to notify the user of such inconsistencies.

The proposed platform is general in its scope, and is in principle able to process any kind of data. However, in order to succeed, any business requires a clearer strategy and focus. Genialis was created as a spin-off of the Bioinformatics Laboratory at the Faculty of Computer and Information Science, University of Ljubljana. Consequently, the platform's primary focus was the processing of data related to bioinformatics. The Bioinformatics Laboratory has previously developed two web solutions: PIPA[10], a platform for next-generation sequence analytics, and dictyExpress [28], a gene expression analytics platform. The platform that Genialis developed was envisioned as the new generation and expansion of the two previous solutions.

While PIPA and dictyExpress served as inspiration, the actual planning and later development of the platform we are describing in this Thesis started from scratch. The basic idea was an application with a multi-layered, loosely coupled architecture. Each part of the system should be independent and interchangeable, all communication between parts should be done through application programming interfaces – APIs. The core layers were the web application, the application server, the persistence layer, and the processing layer. The technologies for the implementation were already chosen for some layers while others, especially the client facing web application, required additional research. We first conducted a survey of JavaScript frameworks for building web applications (the details fir which can be found Section 4.2.1). After choosing the technological solutions, we started planning the architecture. The database outline was constructed and the APIs for interconnecting the parts were devised. Finally, implementation began as an iterative pro-

[10] `pipa.biolab.si`

cess, first creating a very basic solution that was gradually upgraded and expanded with additional features.

Thesis outline

We continue with the description of the proposed architecture in Chapter 2 which looks at the theoretical concepts and architectural practices we leaned on during development. The first part of the chapter details the concepts related to multi-layered software architecture and the second part explains dataflow programming. Chapter 3 is a survey of existing and related solutions where both relevant desktop and web applications are examined. Chapter 4 describes how the theoretical concepts and ideas were put into practice when implementing the platform. It begins with the development goals and architecture, continues with more specific solutions for each part of the application, and finally explains the implementation details of the key parts. Chapter 5 concludes the Thesis by reflecting on the work that was done and considering further features that can still be implemented in the future.

Chapter 2

Theoretical background

2.1 Architecture

The client-server architecture provides the scaffold of any distributed application that communicates through the Internet. It naturally evolved from the times of centralised computing to how (simple) client-server communication in web browsers works today. With the advent of cloud computing, the architecture of applications has become much more complicated, but in the most general terms it still adheres to the old client-server separation of a service provider and a service consumer. Best design practices have emerged and the standard for modern web applications is a multi-layered architecture. Most commonly there are three layers: the presentation layer, the application (also called business, business logic, logic, or just the middle) layer, and the persistence (or data) layer. There is some confusion regarding the usage of the terms layer and tier when describing an architecture, the words often being interchanged. Fowler [12] explains that the difference between the terms is in their implication – tier implies a physical separation (different machine). A multi-layered application is thus not necessarily multi-tiered, the application and persistence layer can run on the same physical machine.

The presentation layer of a web application is the interface through which a user interacts with the application; it displays information and results,

handles users' actions, and transforms these actions into requests to the application layer when required. The presentation layer is the client portion of the client-server architecture and because of that, it is also often referred to as the *front-end*; the same analogy applies to the rest of the layers resulting in the term *back-end* being used to refer to the server part of the application. The persistence layer concerns itself with data storage and the interfaces for accessing the stored data. Between them is the application layer which handles communication between the other two layers as well as all the functionality of the web application. The application layer is usually the most complex of the layers and often ends up being multi-layered itself. Because of this, the three-layered architecture is sometimes referred to as multi- or n-layered to account for the additional separation. Additional layers inside the application layer can be, for example, the service, business logic/domain, data access, and the processing layers. This separation can once again either be logical or physical (different machines) as well, thus resulting in additional tiers. Even if some components of the application layer end up logically separate, other parts, like logging or security measures, are vertically integrated throughout the layers to ensure proper functionality. An example of such an architecture is shown in Figure 2.1.

The service layer represents the middle-ware interface to which the presentation layer connects. Such an interface substitutes the typical request-response communication between the client and the server, applying an additional level of abstraction between the user interface and the application layer. The benefit of such an abstraction is that various kinds of clients familiar with the API of the service can access it – instead of just the user interface, the service can also be accessed by other applications. This concept is further explained in Section 2.1.6. The business logic or domain layer is an abstraction of the concepts, rules, and functionality of the problem domain, encoding the possibilities of what can be done with the data. Evans [10] calls it "the heart of business software". The data access layer is the bottom middle-ware, responsible for the communication with the data layer. It

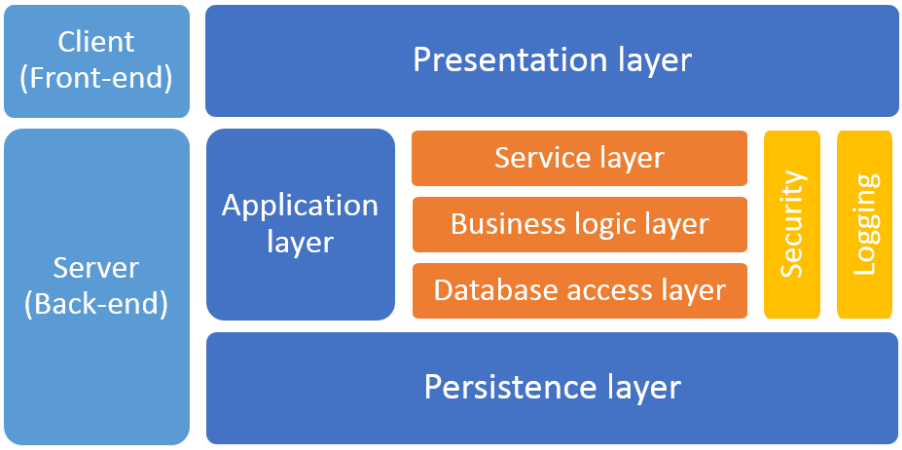


Figure 2.1: A multi-layered architecture with a separation between the front- and back-end, the three typically used layers, and a more detailed breakdown of the application layer. The security and logging blocks are examples of modules that are accessible throughout the inner layers of the application layer.

typically contains an object-relational mapping system.

Lastly, there is the processing layer, although the processing 'part' would probably be more accurate. Typical architecture descriptions do not involve a discussion of this part but we believe it is an important one. We understand the processing layer as the one handling the scheduling of jobs and management of the processing units – workers – that perform the tasks. If the jobs ordered by a user through the interface are long-running and numerous, they require a dedicated system for their execution. This becomes important when the scalability of the application is considered and is further explored in Section 2.1.2.

Most of the benefits of a multi-layered architecture stem from the *separation of concerns* principle. Separation of concerns is one of the core software architecture design paradigms and aims to split the whole into parts that address different concerns. When describing this technique of ordering one's thoughts, Dijkstra [9] explains concerns as a way of focusing on one aspect of the whole that can be viewed as independent from the rest. The principle

can be put into practice by utilising modular design.

2.1.1 Modular design

Modular design is attributed to David Parnas [25], who introduced it as a design principle that shortens development time along with improving the flexibility and comprehensibility of the product. The key to a good modular design of a system is the correct utilisation of *information hiding*. Modules should be isolated parts of the system based on their functionality and they should separate the interface from the implementation – hiding internal data and functions while exposing the functions that other modules can access as an interface. This way, if the implementation of the module has to be modified, it will not break the system. Other modules access only the interface which should remain stable even if how the functionality is implemented changes completely.

Stevens et al. [33] use two additional terms to describe the modularity of a system – *cohesion* and *coupling*. Cohesion describes the degree to which the elements of a module are bound together; each module should be highly cohesive, both in terms of its implementation and the functions it can perform. For example, a part of an application with multiple responsibilities can be split into multiple modules with high cohesion that handle one of those responsibilities. Coupling describes the relationship between modules, examining how interconnected and (in)dependent they are. Modular design strives towards loosely coupled modules – a module should use little or no knowledge of the implementation of other modules. Ideally, two modules with the same interface and completely different implementations should be interchangeable without breaking the system.

Examples of modular design can be found on all levels of software development, from object-oriented programming paradigms to how software libraries and modules are packaged and developed. On the higher levels, it separates parts of a system into logical units, an example of which is the multi-layered design mentioned above. By utilising the principles of modular design and

therefore developing loosely coupled, highly cohesive modules for each part of our multi-layered design solution, we should end up with modules that communicate through application programming interfaces. An API is the stable interface of a module through which its functionality is accessed; an API can also be understood as a specification of the functionality. Development can happen in both directions – implementing the functionality of a specific interface as well as exposing the capabilities of a piece of software by defining its interface. Sometimes APIs can be standardised, for example the *POSIX* family of standards which describe the interfaces of Unix operating systems or *SOAP* and *REST* service specifications, which are further explained in Section 2.1.6. As mentioned above, modules with equivalent APIs should be interchangeable, regardless of their implementations.

An examination of the benefits of modular design can begin by looking at Parnas' goals and how they are achieved. Development time is shortened because different independent modules can be developed simultaneously, after the common interfaces are agreed upon. Maintenance is shortened because a change to a module affects just the module and there is no need to modify additional parts of the application; the dreaded ripple-effect of changes affecting increasing portions of an application should not happen with good modular design. Comprehensibility of the whole system is significantly improved compared to tightly coupled systems because each module concerns itself with one role instead of functionality being implemented in multiple interconnected parts of the application. Flexibility is also increased. As we have mentioned above, ideally a module should be easily replaceable with a different module with an equivalent API. This becomes important when software eventually becomes outdated or better solutions are found – integrating these into the system is much easier when only the interface has to be compatible.

Module re-usability is another important benefit – self-contained parts of the system can easily be reused elsewhere when their functionality is needed. To make sure the written code performs in the way it is expected, tests are

performed. Specifically, unit (also known as component) testing is usually employed to ensure that each part functions correctly. Modular design on all levels helps keep components succinct, loosely coupled and thus easily testable.

There are, of course, some drawbacks to modular design as well. The biggest is a communication overhead. The amount of overhead is relative to the size and the heterogeneity of a system. Modular applications inside the same domain experience minor overhead through the use of interfaces, while communication in large multi-layered applications, spread across machines and technologies, can have a significant impact on performance. Communication between components based on different technologies often happens through standardised messages. Such messages need to be transported, translated, and interpreted before they can be used (an example of such messaging systems are web services, explained further in Section 2.1.6). Another potential drawback is the so-called *dependency hell* that can occur when using various modules and packages that come with dependencies of their own – the more that are used, the greater the potential for circular and/or conflicting dependencies, which has a detrimental impact on the comprehensibility and maintainability of a project.

The communication overhead is an acceptable drawback when all the benefits of modular design are considered. In cases where different technologies are communicating between each other, some kind of an API will always have to be used. Potential problems with dependencies can be avoided by careful planning or by utilising systems for managing dependencies. In general, modular design incorporates so many of software development best practices that it should be used at any scale, but with large, multi-layered applications, it seems quite necessary.

2.1.2 Scalability

When developing a web platform with as grand a scope as we had, it is very important to keep scalability in mind – the system should be able to handle

anything it may eventually encounter. To discuss scalability, we first need to clarify two important concepts: *big data* and *cloud computing*. Both seem to lack a clear definition, changing and evolving over the years to suit emerging trends. The original definition of big data can be traced back to a research report by Laney in 2001[11] that defined big data as data exhibiting an increase in the three *Vs* – volume, velocity, and variety. Highly varied data, ever increasing in size and scope, was (and still is) being created at unprecedented speeds (velocity does not just refer to the speed of data generation but also to the speed at which such data needs to be processed). Such (big) data could not be handled by what was then conventional data processing; it required new approaches and ideas. This was the basis of the currently most commonly accepted definition of big data – data that is too large and complex to be processed with standard approaches [31], with 'processing' referring to anything to do with data, be it capturing, storage, analysis, etc.

Cloud computing is another similarly ambiguous term. The authors of a paper titled "Demystifying Cloud Computing" [19] write that they uncovered 22 distinct definitions of cloud computing and opted for one taken from Gartner, Inc. which states that is is "a style of computing where massively scalable IT-enabled capabilities are delivered "as a service" to external customers using Internet technologies". This is an accurate definition of cloud computing but it mostly refers to the so-called public cloud, where infrastructure, platforms, and software are offered as a service. The other option it does not include is the possibility of building a private cloud where the same massively scalable IT-enabled capabilities are harnessed by only the company that built it.

X-as-a-Service (XaaS) or *anything* as a service, is the model that describes how cloud computing providers offer their products. In exchange for a fee (or sometimes free of charge) the service is offered through cloud clients. Figure 2.2 shows the hierarchy of these services. The clients are devices or

[11] `blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf`

software receiving the service, typically web browsers or mobile applications, but also various other thin clients. The functionality, or the level of control, a user is granted through the service corresponds to the level of service the user is using. The following explanations are based on the NIST definitions[12]. Infrastructure-as-a-Service (IaaS) usually offers virtualised fundamental computing resources (processing power, storage, networks,...). The user is granted full or a high level of control over the components but not over the underlying cloud infrastructure – they can create and manage their own virtual machines. The next level is Platform-as-a-Service (PaaS) where the user is provided a platform with resources on which they can run their applications (if the execution of the applications is supported by the service). The user thus has control over the application and can manage how it is run, yet has no access to how the infrastructure is setup and run – the user can usually manage the deployment and configuration of the application. The big benefit of PaaS is the reduction in complexity when compared to IaaS, as users can focus on the application and let the service provider solve the infrastructure portion. Software developers choose between IaaS and PaaS depending on the level of control they require over their application; interestingly, PaaS providers usually do not build their own clouds, they use IaaS to offer their services. Software-as-a-Service (SaaS) grants end users the usage of applications deployed on cloud infrastructure. It can be understood as a model of software distribution and licensing, where the software is centrally hosted and accessed through clients by subscribing users/customers. Developers use such a model because of the numerous benefits it offers when compared to regular software distribution through binary files, these include: control over software updates, access to user data and behaviour, cross-platform and device compatibility. The end-users also benefit from some of the listed advantages, but others are potentially big drawbacks of the model, especially the fact that end-users often relinquish control of their data, thus facing potential privacy and security issues; and a complete copy of the software

[12] csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

is never owned by the user. Mitigating these drawbacks – particularly the security and privacy aspects – is one of the keys to successfully developing SaaS products.

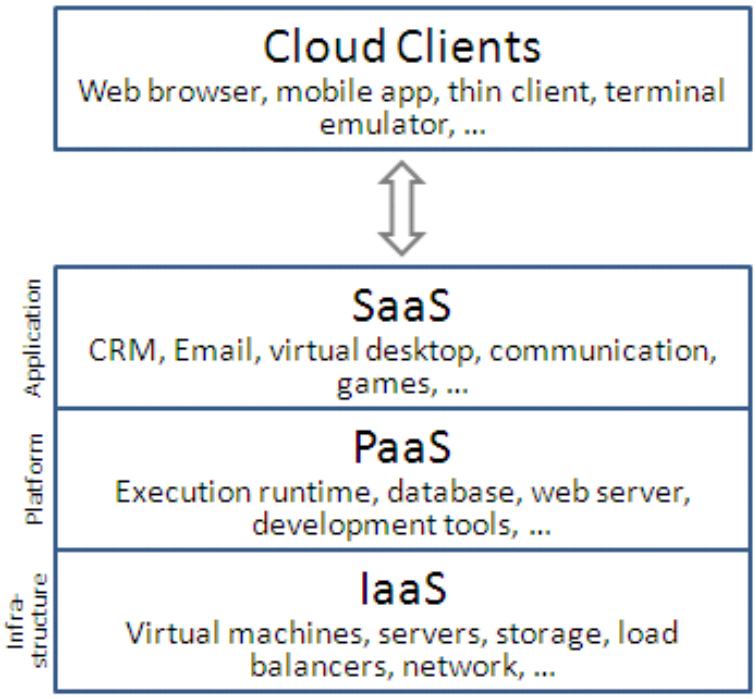


Figure 2.2: The hierarchy of cloud computing services.

Lastly, let us touch upon scalability. Bondi [5] defines it as "the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement". In general, there are two ways a system can add resources: scaling horizontally or scaling vertically [21]. A system running on a computational node can be scaled up by increasing the power of the node, for example by adding additional memory or CPU power to the node. Scaling out, or horizontally, means adding additional computational nodes. Of course, a system needs to be designed with such horizontal scalability in mind to properly utilise additional nodes. By adhering to the above-mentioned modular design, such horizontally scalable applications can be written.

Where do the three definitions come together? Cloud computing is the

key to the achieving affordable, flexible scalability of a web platform. IaaS and PaaS can be used to deploy an application in a manner that allows for easy horizontal scaling simply by adding additional instances of whichever module or modules the system needs. A good example of such scaling can be found in the above-mentioned processing part of the hypothetical multi-layered application. When the job scheduler is given many jobs to handle – an amount that would take too long to process with the current number of workers – additional worker nodes can be temporarily spawned on the cloud infrastructure to handle the surge in jobs and once the work is processed the additional worker nodes are destroyed. With a major surge of job requests, even more additional job schedulers can be spawned – the same principle applies to all parts of the application.

2.1.3 Data storage

A modern web platform for data processing should be able to store and process any and all kinds of (big) data. The most commonly used database management systems (DBMS) have been (and still are, for most use cases) relational databases. Along with the rise of cloud computing and big data use-cases have appeared that could not be solved with relational databases adhering to the ACID standard of transactions. The acronym ACID was first used by Haerder and Reuter [16] and stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*, although the definition of these properties is attributed to Gray [15]. The properties describe how transactions within a database need to be processed to ensure reliability, accuracy, and efficiency. Relational databases were able to offer such transactions and the solution was satisfactory until the need for massive scaling of databases arrived. Classical relational databases have come up against the Brewer's CAP theorem [7] which states that no distributed system can simultaneously guarantee *Consistency*, *Availability* and *Partition tolerance*. A consistent system ensures all nodes have the same data at all times, the guarantee of availability requires that every request to any (non-failing) node must result in a response, and

partition tolerance guarantees that the system will continue to operate even if parts of it fail. Classical relational databases sacrifice partition tolerance to ensure consistency and availability, thus sacrificing horizontal scalability. In a 2012 update [6], the author of the CAP theorem adds that the three categories are not binary and should be understood as continuous and also that the two-out-of-three rule can be potentially misleading if understood categorically instead of with a nuanced approach. New solutions have been gathered under the umbrella term NoSQL (*Not only SQL*) – SQL is the most commonly used query language in relational databases – though a more accurate term for it would be non-relational databases.

No definitive categorisation of the NoSQL databases exists but they are generally grouped based on the data model they use to represent information: key-value databases, document databases, graph databases, and column databases. Additionally, there are two general directions that NoSQL databases are able to choose from in order to gain partition tolerance – either sacrificing consistency or availability. If the DBMS sacrifices availability, it can still perform ACID transactions but usually has to limit them to those that can be consistent, refusing others. The other option is giving up on consistency, and responding to all requests, but potentially returning outdated data. To describe this opposite end of an ACID transaction, another acronym was created – BASE [27], standing for *Basically Available*, *Soft state*, and *Eventual consistency*. The most important one of the three is eventual consistency – in a distributed system, one node receives new data and eventually propagates it through the other nodes, thus ensuring they eventually all have the same information (this is a very simple example, glossing over potential conflicts and security measures). An additional important difference between NoSQL and relational databases is that the former usually gives up on complex data schemata and efficient complex queries that relational databases are able to do, in order to gain the benefits described above.

2.1.4 Model-View-Controller

Model-View-Controller (MVC) is an architectural pattern for implementing user interfaces. It was described by Krasner and Pope [20] as a way of separating how data is presented inside the application (model), how the user sees the data (view), and how the user interacts with the data (controller); Figure 2.3 shows a simple MVC diagram. This separation increases the modularity and portability of interface components.

The model portion of the application is an implementation of the problem domain of the application – what data represents a certain state and how it can be changed. The view is a presentation of the model (or parts of it) as information in the user interface. The controller defines the possible interactions a user can input, transforming these inputs into commands. The controller sends commands to the model to cause some change in the model or commands the view to change how the model is represented. Changes in the model cause the view to update the representation.

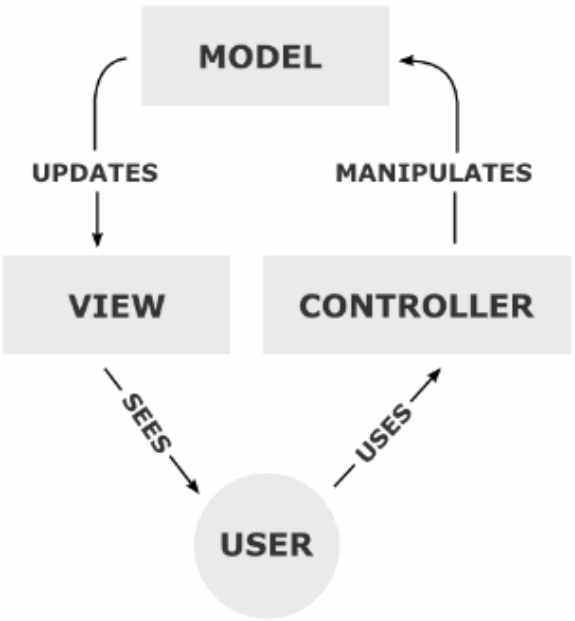


Figure 2.3: A simple diagram of the MVC architectural pattern.

The same model can have multiple views and controllers defined for it. A simple example is a hypothetical drawing application. The state of the drawing and possible manipulations of it are defined in the model. The view of the drawing represents its canvas but could also be a zoomed-in view of a portion of the canvas – a different view of the same underlying model. The controller converts the mouse inputs (e.g. drawing a circle) into commands for the model and the changes of the state of the drawing end up reflected in the view. Because of the modular nature of the architecture, adding support for a pen and tablet input device is relatively simple – only the controller needs to be updated.

Since the specification of the MVC pattern, various implementations of it have ended up working off the initial idea to, creating derivative patterns such as *Model-View-Adapter*, *Model-View-ViewModel*, *Model-View-Presenter*, etc. The whole class of such patterns are commonly referred to as *MV** because they mostly differ in how they define what the original pattern refers to as the controller, as well as the connections between the three parts.

2.1.5 Object-relational mapping

Query languages are typically used to communicate between an application and a database and the most commonly used query language is *SQL*. When data needs to be fetched, an SQL query is constructed and executed, the returned data entries are then usually converted to objects (in object oriented programming languages) to be used in the application. *Object-relational mapping* (ORM) exists as an abstraction of this process, automating the saving, query generation, fetching, and the conversion into objects so that programmers only work with objects.

ORM frameworks are a very welcome level of abstraction, significantly cutting down on the amount of code that needs to be written and allowing the programmers to stay within the object oriented world and not having to write SQL at all. The ORM system essentially handles everything related to the database, starting with the creation of tables that match the objects.

Additionally, the database solution can be exchanged for another compatible one without major changes to the code of the application, usually just by switching the adaptor handling the communication to a specific database solution.

ORM frameworks are often criticised for being bloated, inefficient and guilty of obfuscation, the gap between object-oriented and relational approaches – the so called *impedance mismatch* – is difficult to bridge as well [29]. The conversions between the objects and data add overhead, the generated queries can be inefficient, and the programmer is often not aware of what exactly happens. However, when programming in object oriented languages, the data from the database will eventually have to be transformed into objects, meaning the programmer will end up writing some sort of mapping between objects and the relations inside the database – so why not just use an ORM framework? Most ORM frameworks support *manually* written queries for when performance is critical and at the same time offer plenty of benefits. Object-relational mapping is supported by all commonly used object oriented languages with various modules/packages/frameworks, and although SQL was often mentioned, ORM solutions for NoSQL databases also exist.

2.1.6 Representational state transfer

Service oriented architecture (SOA) is an architectural pattern where communication between components happens through services, typically over a network – one component of a system provides the service and another consumes it. Because the underlying platforms of services can be very diverse, a standardised way of defining services and the communication between providers and consumers was needed. Solutions were proposed and the *Simple Object Access Protocol* (SOAP) was chosen as the standard protocol for communication[13]. SOAP wraps messages to be exchanged in an envelope and transfers them through common transport protocols like HTTP or SMTP.

[13] `www.w3.org/TR/soap12`

SOAP was designed for expansion, and many additional standards and protocols have been developed as solutions to the problems and shortcomings of the original protocol. Examples of such solutions are *WS-Security*, developed to provide end-to-end security, and *WS-Atomic Transaction*, developed to ensure the atomicity of transactions. Web services related specifications are often referred to by using the term *WS-**.

SOAP, along with the whole WS-* stack, is perceived to be complicated and unnecessarily complex when the goal is exposing a relatively simple service [26]. An alternative solution is *representational state transfer* (REST, also ReST), which was first described by Fielding and Taylor [11]. APIs developed according to this style are called *RESTful APIs*. REST is not a standard but an architectural style, although most implementations do follow a set of rules. Compared to SOAP, which packages data inside XML envelopes, REST uses the capabilities of HTTP and its standard methods (such as GET, PUT, POST, DELETE) to expose a service's resources. Without the XML envelope, RESTful resources can return data in a form that can immediately be used, most often objects in the JSON notation – the output can match the intended usage.

Choosing an appropriate solution mostly comes down to intended usage. Developers of simple APIs that are intended to be used through HTTP only usually opt for RESTful APIs, while more complicated solutions requiring advanced features across multiple platforms and communication channels use the WS-* solutions and SOAP [26].

2.2 Dataflow programming

Dataflow programming is a programming paradigm that models programs as directed graphs, where data *flows* between nodes that represent operations [17, 32]. It represents a shift away from the traditional thinking about programs as a sequence of instructions executed one by one by a von Neumann processor. In the directed dataflow graph, an operation can be exe-

cuted as soon as its inputs are available; this allows instructions independent of each other to be executed in parallel. Because of this separate processing model, we can speak of the *dataflow architecture*, which seeks to implement the paradigm as a computer architecture, and *dataflow programming*, which tries to achieve the goals of the paradigm through software.

The discussion about the hardware implementations, the history of dataflow programming, and the details of its implementation are beyond the scope of this work. Interested readers are referred to the article by Johnston et al. [17]. We will take a look at the basic dataflow execution model as it is the conceptual basis of later research.

2.2.1 Dataflow execution model

As we have already stated, a program in the dataflow paradigm is represented by a directed graph where the nodes represent primitive instructions, such as arithmetic or comparison operations; the following paragraphs are a paraphrase of the explanation of the pure dataflow model in [17]. The directed arcs of the graph represent data dependencies between nodes, *inputs* flow towards the node and *outputs* flow from it. The data flows between nodes in the form of data tokens and an arc functions as a first-in, first-out queue for the tokens. At the beginning of the program, the so-called *activation nodes* place data tokens onto initial arcs and with that the network can begin processing. When a node has the specified set of input tokens available through its arcs (a *firing set*) it is said to be *fireable*. Such a node is executed at an undefined time after it becomes fireable, removing its input tokens from the queues, processing then, and placing new data tokens on some or all of its output arcs. After this, the node waits until it becomes fireable again. This is the key difference from the von Neumann execution model, where an instruction is only executed when the program counter reaches it, even if it could have been executed earlier. Dataflow processing supports parallel execution at the instruction level.

Figure 2.4 shows a simple example of a program (a) and its dataflow

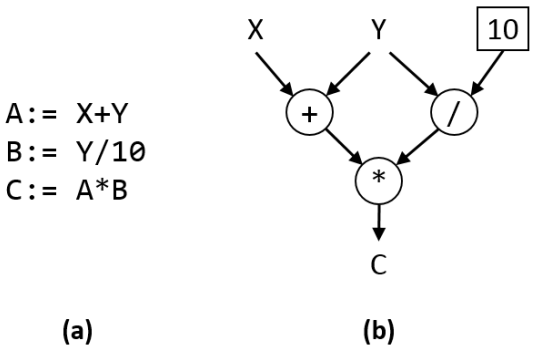


Figure 2.4: An example of a simple program (a) and its dataflow equivalent (b), shown as proposed by Johnston et al. [17].

equivalent (b). The capital letters represent variables and the number in the box represents a hard-coded value. The two arcs coming out of variable Y represent a duplication of the value of the variable, the copies ending up in their respective nodes. Program (a) requires three time units to complete but the dataflow version can be completed in two time units because the first level of operations can be completed in parallel. The dataflow nodes are *functional*, meaning they do not modify the input data and have no side effects on other nodes.

This is the basic description of the dataflow execution model at the instruction level, the finest possible granularity. In the 1990s, researchers started realising that the von Neumann and the dataflow architectures are not mutually exclusive, but are instead two extremes among the possible architectures [24]. Fine-grained dataflow was seen as a multithreaded architecture where each machine-level instruction is executed in its own thread. This shift in perception moved research towards the so-called *hybrid dataflow*, or dataflow of varying granularity. With this we move onwards toward a more macro perspective.

2.2.2 Flow-based programming

Flow-based programming (FBP) was invented by Morrison [22, 23] in the early 1970s. In the past, it used to be referred to as *dataflow programming* (as well as by other names) but it was renamed to FBP to differentiate it from the research being done in the dataflow field and because dataflow is more general in scope.

FBP views an application as a network of asynchronous processes, communicating by exchanging streams of structured data called information packets (IPs). As with dataflow, the focus is on the data being exchanged and the transformations applied to it. The network is defined externally as a list of connections, separate from the processes. These connections are interpreted and executed by a piece of software that Morrison calls the *scheduler*. Before we continue with a more in-depth look into FBP, it is important to understand that Morrison is describing both the paradigms of FBP as well as the manner of their implementation. We are more interested in the concepts and will be simplifying some parts that are too focused on the implementation.

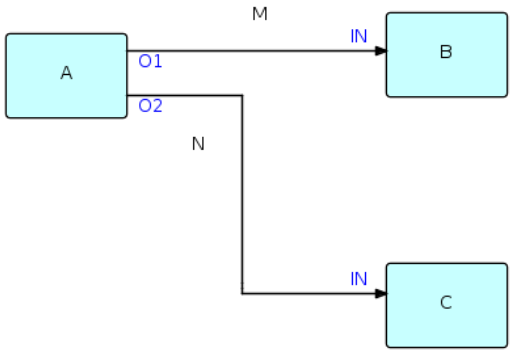


Figure 2.5: A simple example of a FBP diagram[14].

In essence, FBP splits the development of an application into two parts

[14]Image created by Morrison and shared on Wikipedia (`en.wikipedia.org/wiki/Flow-based_programming`).

– designing the network diagram and implementing the components (nodes, processes). The components are treated as functional black boxes with inputs and outputs and the network describes the connections between them. Figure 2.5 is an example of a very simple network definition. A, B, and C are the black box processes, $O1$, $O2$ and the two IN represent ports through which connections M and N connect the components. Morrison notes that it is possible for the B and C boxes to be executing the same code, and because of that, each component must be independent from the rest, with its own local storage and control blocks. For the same reason, shared port names are not a problem because they are relevant only in the context of a component. The connections are referred to as *bounded buffers* with a fixed capacity of IPs; if the buffer is full, the process feeding stops, and when the buffer is empty, the process is suspended. This very high-level overview should be enough to show how the components are defined and the similarity to the classical dataflow approach explained above. Readers that are interested in further details should refer to Morrison's book [22].

FBP can be implemented in various programming languages, and Morrison provides some example implementations on his website[15]. The implementation allows programmers to write the high-level components and determine how they are interconnected in conventional imperative programming languages. Morrison notes that if the scheduler is written in a sufficiently low-level language, components written in different programming languages can be used within a single network.

FBP exhibits the properties of modular design on a relatively low level, connecting together functional chunks of code to create parts of, and ultimately complete, applications. Building an application with the components naturally leads to loose coupling because the components only exchange data (and signals) between themselves. It also leads to the possibility of component re-use, easier maintenance, and a straightforward way to utilise horizontal scaling as well. Morrison also states that FBP is very useful for rapid

[15] `www.jpaulmorrison.com/fbp`

prototyping, first using simulated components that can later be swapped with components containing real process logic. It also never loses sight of the big picture, offering a macro overview of the functions of each component/module. The network schema as a medium eases the communication between everyone involved – from designers and developers to users and management.

2.2.3 Visual programming languages

In the past, dataflow programming used diagrams (also called schemata or just graphs) as a visual aid to facilitate understanding. The concept of the *data flow diagram* as the flow of data inside an information system can be traced back to the authors [33] of structured design that we discussed at the beginning of the Section 2.1.1 on modularity. Because the diagrams were so useful for understanding and reasoning about systems, it is no surprise that there were also incentives to use them as a means of constructing systems; this resulted in the invention of *visual programming languages* (VPLs). VPLs also exist at varying levels of abstraction, from very simple programming constructs to whole modules being connected together into a system.

Johnston et al. [17] present a well referenced historical review of visual (as well as text-based) dataflow programming languages. It includes an assessment of the state of the art and identifies areas that are still in need of development, but the overview focuses mostly on dataflow execution paradigms and implementation. We will examine their conclusions on data visual programming environments.

Developers face a paradigm shift when switching between the design phase and the coding phase [1]. When designing, they are naturally inclined to use the dataflow approach but the coding is typically imperative. Visual programming languages blend the design and coding phases into a seamless process. As stated before, the graphical representation is helpful on many levels, Baroth and Hartsough [1] cite examples of dramatic gains in productivity as a result of the improved communication between the customer, developer,

and the computer due to the usage of visual programming tools. The level of success depends not only on the VPLs but also on the development environment in which they are used – the distinction between the environment and language is difficult to make because of the graphical nature of VPLs. Well-designed development environments with animated executions of programs and the ability to seamlessly reason on multiple levels of abstraction can greatly benefit both the development and the design phase, especially.

Finally, the most important benefit in the context of this work is the fact that visual programming languages are intuitive to people with little to no programming experience. There are plenty of successful tools using visual dataflow interfaces as part of their interfaces, and many educational tools that teach programming and robotics to children also employ visual programming interfaces. For example, *MathWorks Simulink*[16] is a graphical programming environment for modelling and simulating dynamic systems, *Scratch*[17] is a free educational VPL, and the *LEGO Mindstorms* robots[18] can be programmed with a VPL. Dataflow interfaces are ideal when the end user requires complex operations but does not have the programming knowledge to implement them.

2.2.4 Triggers

The concept of triggers was touched upon in this Thesis when discussing the dataflow execution model. A node becomes fireable when all its inputs are available and is then processed at some undetermined time. It is important to note that the execution does not necessarily happen immediately after the last input arrives; this is the difference between a task being *enabled* – fulfilling all the prerequisites for execution – and being executed [34]. The external condition that leads to the execution is the *trigger*. Actual execution is in the case of the dataflow execution model handled by the underlying

[16] `www.mathworks.com/products/simulink`

[17] `scratch.mit.edu`

[18] `mindstorms.lego.com`

hardware or software implementation processing the instructions of the node (for example Morrison's scheduler).

Triggers can be understood in two ways [18]: as a verb – an event *triggers* an action if the event's occurrence causes the action to be performed; and as a noun, where a *trigger* is an object (or event) that causes a triggering event. Triggers are mostly discussed in the context of workflow management [18, 34], which is the management and logistics of business processes and activities. Van der Aalst distinguishes between four types of triggers: automatic, user, message, and time triggers. Each represents an external cause that leads to the execution of a task in a workflow. We will move away from the area of workflow management and explain them in the context of a hypothetical web platform on which tasks can be performed. User events are the most straightforward, as the execution of an enabled task is triggered by a human, usually by selecting a possible action through an interface. Time and message events trigger actions periodically and upon the arrival of some sort of a request, respectively. Lastly, there are automatic triggers that trigger the execution of a task as soon as the task is enabled. The dataflow execution model described above can be considered a system with automatic triggering.

When designing the platform, we tried to envision a system wherein automatic processing could be done; ideally, the user would upload the data and a potentially hours-long workflow of linked tasks could be automatically executed. The problem, however, is that the automatic system has to be aligned with what the user wants. Because of this, we envisioned triggers as a set of rules/filters that the user can specify for a task. If potential input data meets the conditions expressed in the pre-set rules, it is automatically processed. This way, the whole workflow can be set up to run automatically upon the arrival of the appropriate data. The details and challenges of implementing such a system are explored in Section 4.3.2.

Chapter 3

Related work

In this Chapter we will examine existing software solutions that are similar or related to the platform we have developed. Some (*PIPA* 3.1, *dictyExpress* 3.2, and *Orange* 3.3) were chosen because they were highly related and served, in part, as inspiration for the platform. These three solutions were developed by the Bioinformatics Laboratory at the Faculty of Computer and Information Science. Others were chosen because they either provide solutions for data analytics and data visualisation or are related to the concepts of dataflow programming. Furthermore, because the domain field for the platform was primarily bioinformatics, solutions from there were considered as well.

3.1 PIPA

PIPA[19] is a web application for managing and analysing *next generation sequencing* (NGS) data. The application supports data storage and processing with a multi-layered client-server architecture – the users access the application through a web browser client. The application fetches data from the server, displaying relevant information in the interface; data processing requests are also handled by the back-end server. Users can annotate the data entries based on pre-set templates, adding all the information they find

[19] `pipa.biolab.si`

relevant. In addition to storing, managing, and processing data the application also supports data visualisation; for example, it uses *JBrowse* [30] to interactively visualise genomes.

One of the most significant limitations of the application is that it is written in *Flash* (Adobe Flash), a once widely used solution for developing rich web applications which was slowly phased out in favour of HTML5 and JavaScript. While the history of Flash is complex, one of the major reasons for its decline can be traced back to Steve Jobs' "Thoughts on Flash"[20], where he describes why Adobe's Flash is not an optimal solution because of its proprietary player as well as issues with reliability, security, and performance. Apple has not supported Flash on its mobile devices and thus Flash was, and is being, slowly phased out throughout the Internet. Maintaining and updating Flash applications in such an environment is a losing battle.

PIPA is important because it was a direct influence for the platform – one of the goals of development was to replace PIPA for the storage and processing of bioinformatics data. Some of PIPA's architecture and solutions were used as inspiration to create a more modern, modular, and scalable product, particularly the multi-layered design with background workers and the open APIs which can be accessed through the web application or through a Python library.

3.2 dictyExpress

dictyExpress [28] is the second of the solutions developed by the Bioinformatics Laboratory in cooperation with Gad Shaulsky's and Adam Kuspa's labs at Baylor College of Medicine – a tool for the visualisation of gene expression experiments of the amoeba belonging to the *Dictyostelium* genus. dictyExpress is also a Flash web application with a database server from which experiment data is retrieved and an analytics server, where the requests for various visualisation combinations are processed and the results cached for

[20] `www.apple.com/hotnews/thoughts-on-flash`

faster retrieval. Figure 3.1 shows the interface of the client application.

The concept of this application is similar to the one used in PIPA, in fact through the exposed interfaces dictyExpress can query PIPA for its data; the technological limitations of Flash apply to dictyExpress as well. Ideally, both applications should run on the same platform, and this is where our solution comes into play – dictyExpress was supposed to be one of the first applications the platform would support.

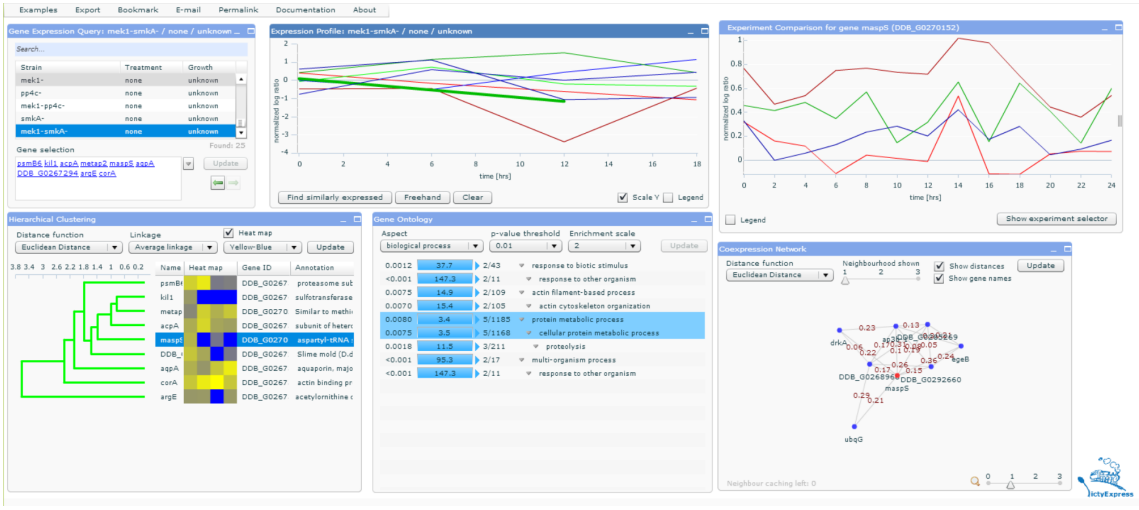


Figure 3.1: dictyExpress' interface for exploring and visualising *Dictyostelium* gene expressions.

3.3 Orange

Orange [8] is a general open source data analysis suite for machine learning, data mining, and data visualisation. It offers a Python scripting library as well as a graphical interface for interaction with the library. The visual programming interface is of particular interest to us because it is a good implementation of the high-level visual dataflow programming. An example of a dataflow schema created with *Orange Canvas* was shown in the Introduction (Figure 1.1). With simple drag and drop operations processing components

(widgets) can be added to the schema, starting with the input *File*. Data flows into the visualisation and processing widgets and updates the components automatically when the preceding widgets or their parameters change. Such an interface enables users to execute relatively complex data analysis workflows without needing to write a single line of code (programmers can use Python scripts as well). Figure 3.2 shows another example of Orange interface, with an open Python interpreter where the results can be analysed programmatically.

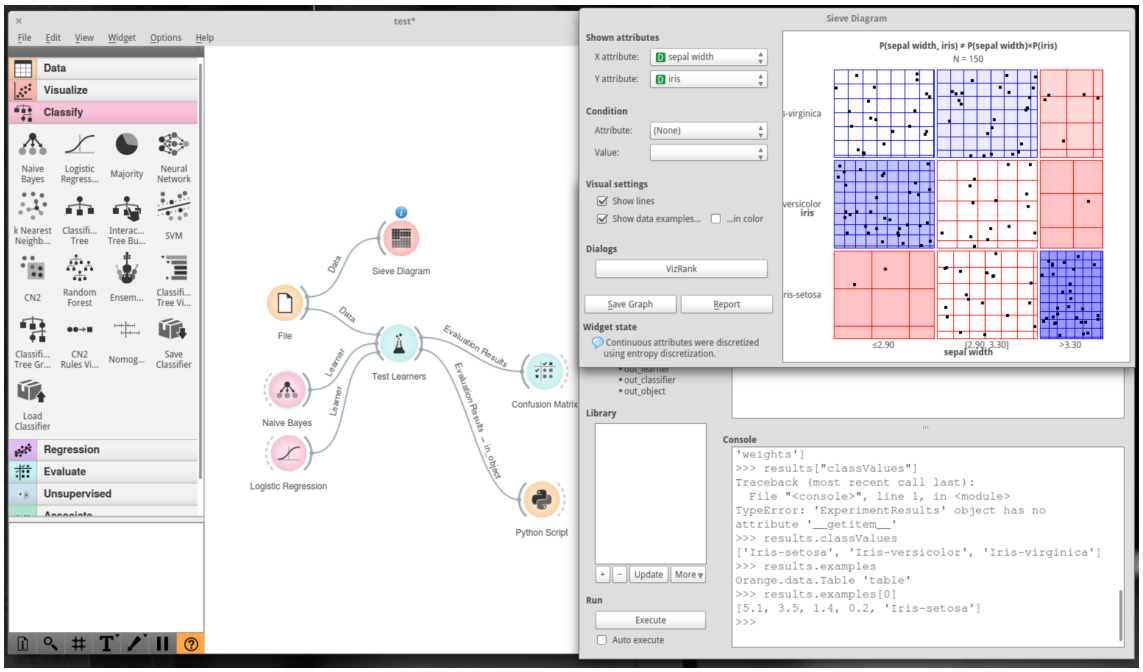


Figure 3.2: A data analysis workflow example in Orange with a Sieve Diagram visualisation and a Python interpreter that uses the results of the prediction.

Orange is not a web application but its visual programming interface is something that the web platform should support to fully implement the dataflow paradigms and reach the widest possible user base.

3.4 KNIME

The *Konstanz Information Miner* (KNIME) [3] is an open source environment for data analysis with a strong visual programming interface. The application is written in Java and does not have an online component; it does, however, offer purchasable extensions to the free open source platform. With these extensions, the platform can become an enterprise level solution with central execution servers and centralised management. Enterprise users work with a desktop client and can remotely execute the workflows they create; results of the executions can also be accessed through web browsers. KNIME is not a web application in the typical sense, as full functionality is not accessible through the web client (although the enterprise edition does have a full SOAP-based API).

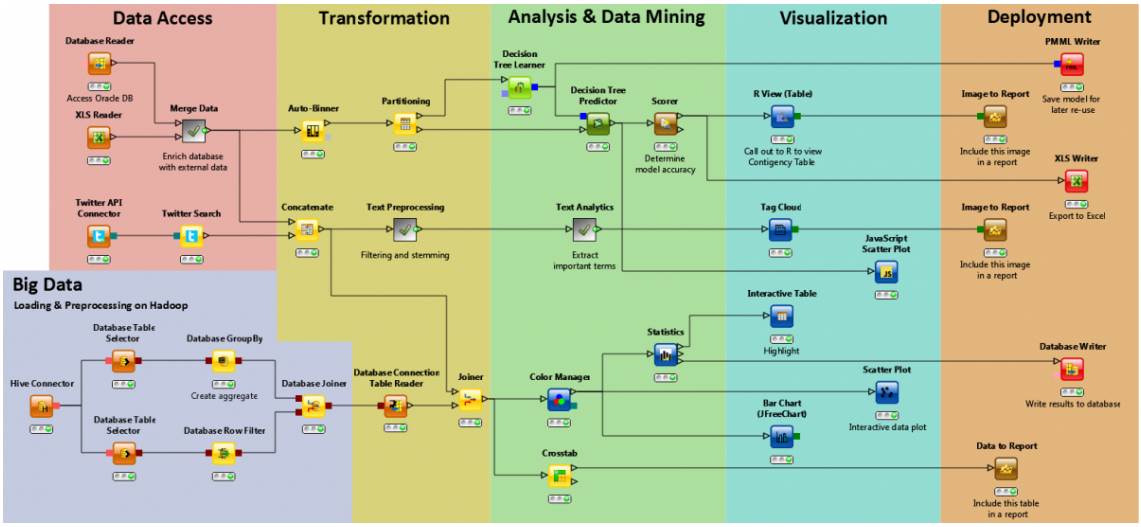


Figure 3.3: An example of an annotated schema from the KNIME platform[22].

As with Orange, the component of interest is the visual interface. It is also a good implementation of dataflow paradigms. A special point of interest is the system of notification semaphores – an intuitive way to inform the platform users of the status of execution (red represents error, orange is

[22]Image available on KNIME's website (`www.knime.org/files/marketingworkflow_2.10.png`).

ready to process, and green indicates that processing has finished).

3.5 noflo.js

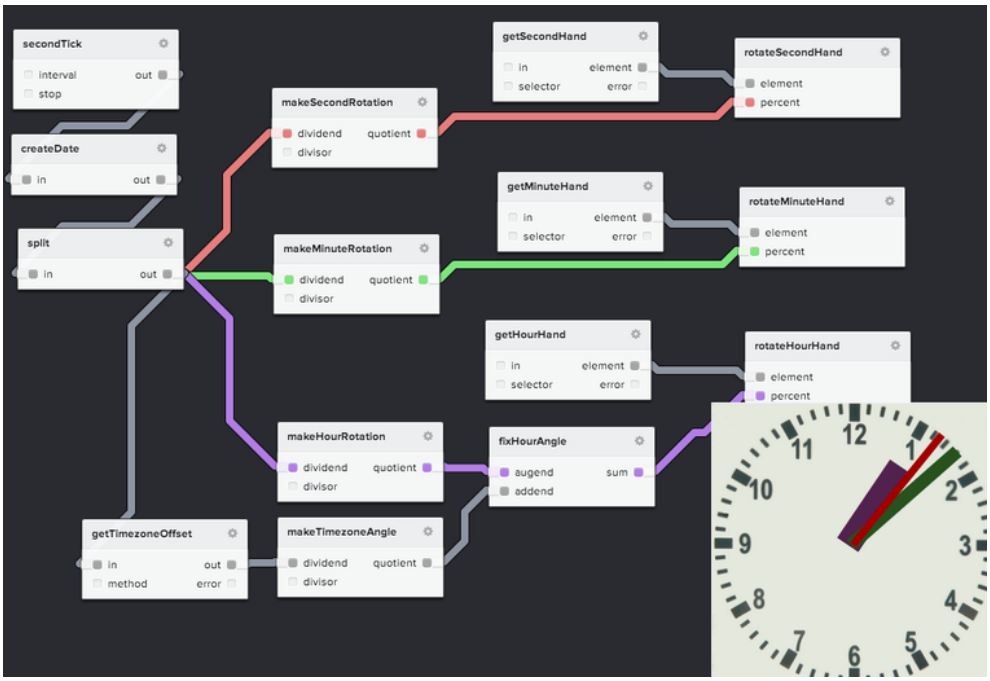


Figure 3.4: An example schema of a clock implementation in noflo.js[23]

Noflo.js[24] is a JavaScript implementation of Morrison's flow-based programming, which was discussed in Section 2.2.2. The project is the result of a Kickstarter crowdfunding campaign[25] and has brought considerable attention to FBP. Morrison notes on his website[26] that the implementation is still not true FBP because it is "bound by the synchronous von Neumann paradigm" by running on *Node.js*. According to Morrison, using a component-based approach and configurable modularity with some visual representation is not

[23] Example available at `noflojs.org/example`.

[24] `noflojs.org`

[25] `kickstarter.com/projects/noflo/noflo-development-environment`

[26] `www.jpaulmorrison.com/fbp/noflo.html`

enough to create a proper FBP implementation, as execution needs to be fully parallel as well.

Nevertheless, *noflo.js* is an interesting solution, especially its FBP inspired web development environment. It is important to note that *noflo*'s intended users are developers – they either write their own components or use pre-made ones to construct applications. Figure 3.4 shows an example of such an application that controls and displays a clock. Each individual component is written in JavaScript (or a language that translates into JavaScript) and they are connected together inside the *noflo* environment. Every second a package is sent from the initial node (named *secondTick*) that triggers the processing throughout the rest of the network.

3.6 Galaxy

In the bioinformatics field, *Galaxy* [4, 14, 13] is the premier open source web-based tool for data processing, supporting a multitude of algorithms. With its visual workflow builders, it provides access to a variety of computational tools to scientists without programming experience. Because it is web-based, it also resolves the sometimes problematic processing environment setup and ensures that all users use the same one. Additionally, it aims to provide transparent and reproducible experiments and analyses which are a very important part of scientific publishing. This is accomplished by storing all the data and processing information server-side. Because it is open-source, anyone can run a Galaxy server privately, which is important – especially when the data being analysed cannot be made public. Galaxy offers free public servers to users as well, enabling those with public data to easily process and share their data and workflows (Figure 3.5).

Galaxy is a very strong player in the field, but according to information gathered by Genialis, its users wish that it were more user-friendly. Another issue that is sometimes mentioned is that Galaxy creates a large amount of data because it cannot directly pipeline the outputs of previous steps to the

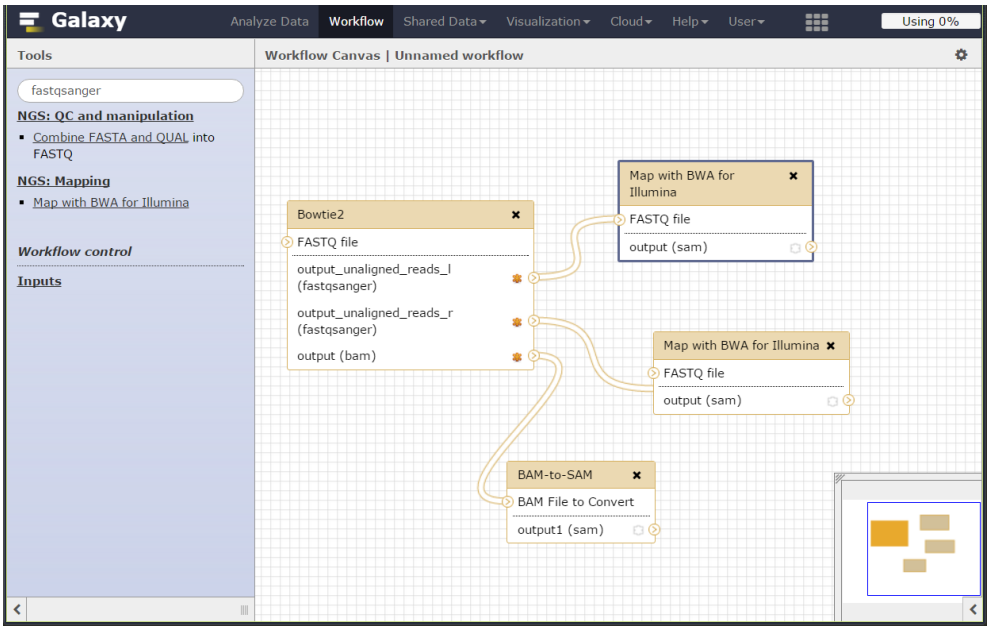


Figure 3.5: An example of Galaxy's workflow editor[27].

inputs of the subsequent ones; instead, it creates files at every step.

3.7 DNANexus

DNANexus[28] is another domain-specific solution, offering a cloud-based platform for processing of bioinformatics data. It sprung up from a Stanford based start-up company and grew through major venture capital funding. Its competitive edge comes from complying with strict security regulations and standards as well as privacy laws. As a cloud-based solution built upon Amazon Web Services, DNANexus offers excellent scalability, user management, reproducible version controlled pipelines, and ease of access to all parts of the pipeline, even to specific nodes in processing clusters.

One of the things DNANexus is missing is a visual dataflow programming interface; currently, their pipelines are implicit by adding steps in a sequence.

[27] Public server accessible at `usegalaxy.org`.

[28] `www.dnanexus.com`

Figure 3.6 shows a simple two stage pipeline, where the outputs of the higher step are used as inputs in the lower one. Additional steps can be added by adding black coloured *apps* (DNANexus' name for processes) and connecting the outputs with inputs.

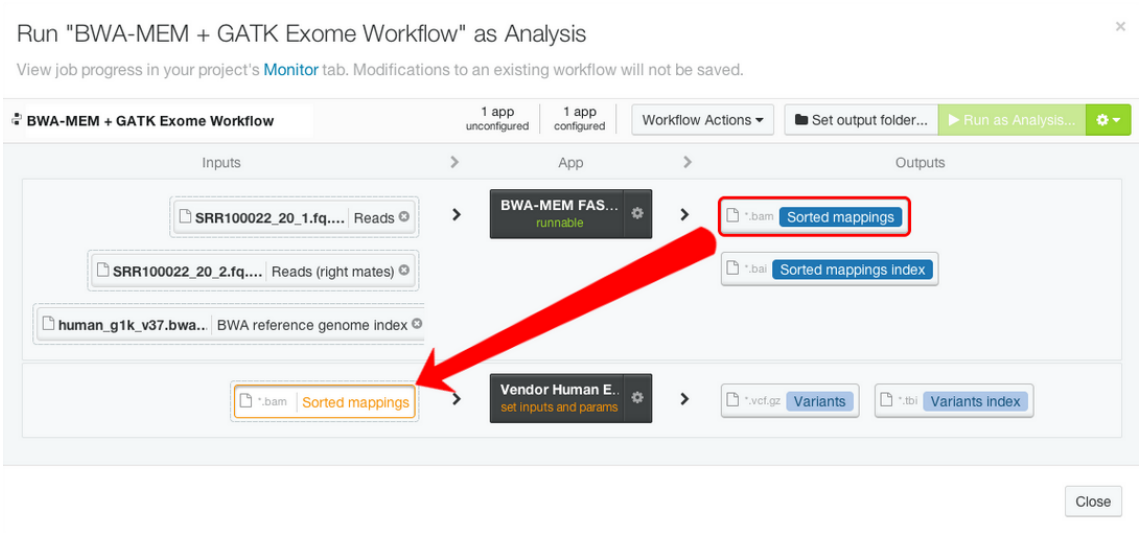


Figure 3.6: An example of a processing pipeline in DNANexus[29].

[29]Image available at `wiki.dnanexus.com/UI/Workflows`.

Chapter 4

Implementation

This Chapter describes the implementation of the platform, beginning with an overview of the development goals, and followed by the overview of the platform's architecture. We also review the technology solutions used in our implementation and the reasoning behind their choices. We also report on some implementation details for selected sections of the resulting application.

4.1 Development goals

As we have already established, the goal of this project was to build a general-purpose web platform for data processing and analysis. The focus was on scalability, extensibility, and ease-of-use. Users should be able to upload all kinds of data files and process them with the tools available on the platform. These tools, or *apps*, as we call them, are added by the developers of the platform as well as by the more proficient users – anyone with sufficient programming skill should be able to develop apps that can run on the platform. These apps can range from simple processing that may only invoke command line functions in the background, to complex visualisations built as web applications. Because the users are given a lot of development freedom, a sufficiently secure processing system is needed through which users can privately run their applications without affecting the processes and data

of other users. Additionally, users who are less proficient and unable to develop their own apps should find it easy to process their data with automated pipelines and ideally with a graphical interface based on visual programming languages.

Before proceeding, we would like to define a few terms that will often appear in the following sections and are the basis for understanding the platform.

Data (also *data entry*) is an annotated piece of information. The platform is capable of accepting all kinds of files but they only become data once they are annotated and stored. The annotation is very important because it holds all the information relevant to the data, such as what the data is, how it was created, and who is its owner. The annotation has fixed and variable portions; the variable part can hold any information the owner decides to add. The data can be used as input to processors to create new data. A data entry can be a single file or contain multiple files, depending on how the particular data type is defined.

A **processor** is a concept very closely related to dataflow programming – it is a single workflow node which can accept data as its input, process the data, and create output data. A process always produces new data and the annotations of the produced output data are closely related to the processor – the data stores the settings with which it was created. Not all processors take data as input; for example, upload/import processors create data entries from raw files instead of other data entries.

An **app** is a client-side application, an interface communicating with the platform's API through the Internet. Apps are expected to be built with JavaScript and HTML but because any technology (capable of sending an HTTP request) can interact with the API, no limitations are imposed. Some apps, especially the basic ones (such as an app for managing projects, data, processes, and triggers), are added by the developers of the platform, but any tech savvy user will be able to create their own app. Apps can be used

to provide advanced interactive visualisations, fetching and processing data via the platform API then rendering it client-side.

A **project** is the basic way of grouping the data – all data entries belongs to at least one project. Such a basic organisational unit was needed because even a single user will most likely work on unrelated tasks which will produce data – grouping such tasks into projects makes sense. Projects also aid in managing permissions – granting someone access to a project grants them access to everything belonging to that project.

Schemata and templates are a way of describing the annotations a data object contains. In the most basic sense, a schema or template represents the description of information. The input, output, and static information is defined/described by the input output, and static schemata. Each object also contains a variable section of information, that information is defined by a template. Different names are used to more clearly separate the origin – the schemata are defined by the processor and cannot be changed, while templates can be changed by users. The processor usually also has a default template set, but the pre-set template can be replaced by end-users, thus changing what kind of information the data entry is annotated with. The reason for the differentiation between the description of the information and the actual information is that the schemata and templates adhere to a set of rules that are used to generate input and display forms in the user interface. The concept is further explained in Section 4.3.3

4.2 Platform architecture

Chapter 2 has alluded to the chosen architecture – a modular multi-layered web application. Figure 4.1 shows an overview of the architecture and the major technological solutions present in each layer. The presentation layer presents the platform through an AngularJS-based application constructed with HTML5, JavaScript, CSS, and Bootstrap. This section will examine

the general structure and goals of each layer, focusing more thoroughly on the parts in which the author was more involved.

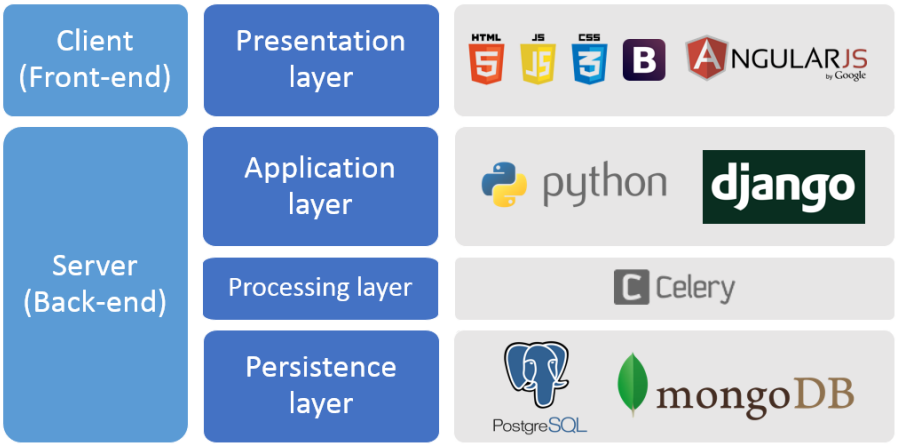


Figure 4.1: An overview of the layers and the relevant technological solutions.

4.2.1 Presentation layer

The presentation layer is the interface of a multi-layered application. In the case of web applications, it is usually accessed through web browsers. This layer is often called the front-end of the application, sometimes also the client-side portion of it. The term application can refer to both the front-end and the platform as a whole; the term *app* (explained above) is an example of that. To avoid confusion, the following sections will use *platform* to refer to the whole application and *application* or *app* to refer to the front-end portion when the platform is being discussed.

Web browsers were initially designed to retrieve and render various static information, but the web did not remain static. Due to the evolution of browsers, client-side interactivity was added through various solutions such as *JavaScript*, *Flash*, *ActiveX*, and *Silverlight*. When the decision was being made on which front-end technology to use, the combination of HTML5 and JavaScript was chosen over the alternatives. The combination is widely supported by web browsers (especially across all mobile devices) because it

does not require the installation of additional run-time libraries. The trend in early 2013 was also turning towards JavaScript web applications – Silverlight never caught on and was not supported on mobile phones, Flash was also beginning its decline, as we explained in Section 3.1.

Selecting the technology was only the first step, because there is a large number of JavaScript solutions available and choosing the appropriate one was quite a challenge. The goal was to find a well-supported, easy to use framework that would ease up the development of the user interface according to the MVC architecture pattern (described in Section 2.1.4). JavaScript offerings generally fall into two categories: libraries and frameworks. Libraries are collections of classes and functions that simplify the code and solve common problems; frameworks are usually more comprehensive architectural solutions that are adapted to suit the needs of the application.

We considered four prospective solutions, examining a multitude of factors, asking: in what way does the solution support MVC interfaces? How mature and well documented is it? Is it easy and intuitive to work with? How easily does it connect with a RESTful API and how well does it support internationalisation? An additional important point is the question of how opinionated is the solution – is it imposing a way of solving the interface problem and do we agree with that solution? After a survey into the possibilities, we narrowed it down to four solutions and ended up examining two libraries – *Backbone.js* and *CanJS* – and two frameworks – *Ember* and *AngularJS*. A test web application was re-written with each of the solutions in order to gain insight and answer our questions.

Backbone.js

Backbone.js[30] is a lightweight library (the smallest of all the tested ones) and was at the time of testing the most popular MV* solution. Its only dependency is *underscore.js*, which it uses for template rendering and for the general utility functionality it provides. The library can be used inside any

[30] `backbonejs.org`

architecture as it is there to provide utility where needed. Unless supplemented by additional libraries, Backbone on its own does not solve all the interface problems, most significantly the issue of data binding between the model and view. It supports one-way data binding; changes in the model are updated in the view but the reverse needs to be coded manually via on-change events. Underscore's templating system, which is used by Backbone is very powerful because it consists of HTML and snippets of JavaScript code that get executed when the template is rendered. This, however, goes against the principle of separation of concerns and ideally we would prefer a more separate system.

Backbone is a good solution when a lightweight, expandable option is needed. It has also been in development for years and is being widely used with a fairly large community.

CanJS

CanJS[31] is another lightweight library that can be expanded to fit the needs of a MV* application. Its main selling points are its extensibility and speed, as it outperforms Backbone and larger frameworks (according to benchmarks on its website). CanJS works with multiple core JavaScript libraries (such as *jQuery* and *Dojo*) out of the box. Its template manager is *embedded.js* and it also mixes JavaScript inside HTML for a highly expressive system, which – like Backbone – lacks a proper separation of concerns. CanJS is a decent lightweight alternative to Backbone but it lacks its maturity and community. In, however, 2013 it was looking promising, having already released a stable 1.0 version in the middle of 2012.

Ember

Ember[32] is a MV* framework and a very opinionated one. The core idea developers should follow is *Convention over Configuration*, favouring Ember's

[31] `canjs.com`

[32] `emberjs.com`

built-in solutions. If the goals of the application align with the framework this results in fast development. Ember uses the *handlebars* templating library for its templates – HTML with embedded expressions that get evaluated once the templates are rendered. Leaving JavaScript out of templates achieves the level of separation of concerns we desire. Compared to libraries, frameworks provide some solutions that considerably cut down on the code that needs to be written, especially due to supporting two-way bindings – changes of the model are reflected in the view and changes to the presented model get propagated back to the model automatically. Ember seemed promising but building an app with it required some modifications of Ember's conventions – these proved very difficult due to a lack of comprehensive documentation. Compared to the others it had not yet reached a stable version and its community was not comparable to the size of the competition's.

AngularJS

HTML is a markup language intended for static documents. Typically, libraries and frameworks manipulate the *Document Object Model* (DOM) to add interactivity to dynamic websites. *AngularJS*[33] is an MV* library that instead opts for a declarative approach, extending the HTML vocabulary. These new HTML tags and properties, called *directives*, get interpreted by Angular's HTML compiler and rendered as DOM elements with specific behaviour attached. This separates the DOM manipulation from the application logic, thus providing a very clear separation of concerns. The consequence of this is a comparatively weaker templating system (it does not allow JavaScript code in HTML), yet the decoupling allows for easier testing, which was one of the primary concerns of Angular's developers. Because of the approach chosen, Angular is a very opinionated framework, to use it the development must be done in *the Angular way*. The paradigm shift results in a relatively steep learning curve and a new vernacular, but the end result is – according to Angular – *what HTML would have been, had it been designed*

[33] `angularjs.org`

for applications.

Angular is not without its drawbacks; its flexibility is touted but getting it to bend becomes increasingly difficult with the rising complexity of the application. The speed is impacted as well, as large DOM trees need to be fully evaluated before the application starts. Additionally, Angular uses the so-called *dirty checking* to detect changes in a model – all values are compared to their previous values, firing change events if the values are different – which is a potential bottleneck when the number of objects increases significantly.

Angular is developed primarily by Google, had an already large and still growing community, was a relatively mature and well documented environment, and provided an excellent MV* solution as long as the limits it imposed were respected.

Selection of the JavaScript solution

We ended up using Angular.js because it met the criteria we set and seemed to have a large enough support by both Google and its community. It took some time to get up to speed with it and we had to circumvent some differences in approaches between different libraries. Ultimately we were satisfied with the choice, the Angular way does work and keeps the code highly modular with custom directives. A basic example of a directive can be seen in Figure 4.2 and Listing 4.1.

For the styling of the solution we turned to *Bootstrap*[34], one of the most popular CSS frameworks. Design was not a priority during development of the first few iterations of the application, which is why Bootstrap was the ideal option, offering quick and easy-to-use constructs to create a decent looking application.

[34] `getbootstrap.com`

```
<!doctype html>
<html ng-app>
  <head>
    <script src="libs/angular.min.js"></script>
  </head>
  <body>
    <div>
      <label>Name:</label>
      <input type="text" ng-model="yourName"
                placeholder="Enter a name here">
      <hr>
      <h1>Hello {{yourName}}!</h1>
    </div> charset test window one two
  </body>
</html>
```

Listing 4.1: Example of a minimal angular application. The application is defined with the keyword **ng-app**, **ng-model** binds the input to the variable *yourName* and **{{**yourName**}}** is set to the current value of the variable. Figure 4.2 shows the rendering of the HTML page, *yourName* set to *test*.

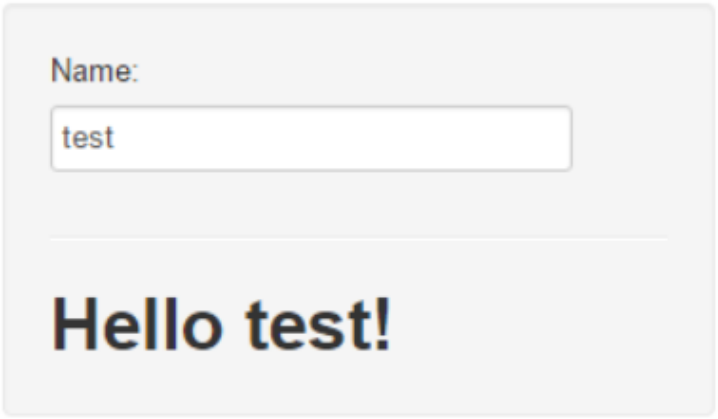


Figure 4.2: The result of a rendered HTML file from Listing 4.1.

4.2.2 Application layer

The application layer is built with *Django*[35], a Python web application framework. Django can be used to build web applications following the MVC pattern where the front-end application fills the role of MVC's views while the models and controllers remain server-side. We use Django as a service – the front-end MVC application views the whole server as the model.

Django was chosen mostly because of familiarity, the team had experience working with it and it was both powerful and extensible enough to suit all our needs. Additionally, it had the relevant libraries available to make sure the stack of technologies worked as we intended it to.

The role of Django is to serve as a container for all the possible front-end apps. It handles the user authentication when users log into the website and also manages their sessions. Django serves the base HTML pages which contain full apps and all their corresponding libraries and styles. Interaction between an app and the server happens through a RESTful web service which exposes the stored objects in a controlled manner. The server also contains the model (object) definitions which are used by Django's ORM system.

RESTful web service

The application layer accesses the presentation layer through an API. Our implementation follows the REST guidelines (explained in Section 2.1.6) and leans on the *tastypie*[36] framework. Tastypie was chosen because it was, at the time, the most fully featured RESTful framework and because it was well integrated with MongoDB[37].

The basic aim of the API is to offer access to the persistence layer, tastypie's term for accessing one of the entities of the persistence layer is a *resource*. The definitions of these resources contain large amounts of busi-

[35] `www.djangoproject.com`

[36] `tastypieapi.org`

[37] The library inter-connecting tastypie and MongoDB is *django-tastypie-mongoengine* (`github.com/wlanslovenija/django-tastypie-mongoengine`).

ness logic, enforcing the limits of what changes can be done to objects and ensuring that no unwanted changes occur. For example, large portions of data entries (such as the inputs and outputs or the date and time of creation) are defined by the process and cannot be changed. Tastypie uses a process it names *hydration* when an object (usually in JSON form, others can be supported as well) is received through the API and then de-serialised from that form into a Python object. The reverse of this process is *dehydration*. Rules can be selectively applied to each field in the object to ensure the behaviour follows the business logic. This provides a level of encapsulation since only the fields relevant to the interface are sent by the server.

Besides the usual Create, Read, Update, and Delete operations, tastypie supports the addition of arbitrary URL patterns and API calls that can execute various functions and procedures on the server. An example of such a function would be accessing all the permissions options a resource supports with a GET request to `platform.domain/api/resource/permissions/`. For authentication security we chose session based authentication, it works closely with Django's sessions, ensuring a user is logged in when API calls are processed. The requests to the API need to provide a valid CSRF[38] token, which is stored by Django into a cookie upon successful login. For authorisation of operations we used a custom system based on permissions, which is explained in more detail in Section 4.3.1.

With a bit of tweaking, tastypie offers a few more interesting features: batch operations, pagination, and querying. Read operations work can retrieve both a single entry or multiple data entries, the other operation can similarly work in batch mode, updating, creating, or deleting multiple data entries with a single request. Simple pagination with page size limits and offsets are easily added as well. Finally, querying inside the objects is supported – the *URL* parameters get translated into queries and executed on the database. Besides simple field matching, nested querying and more advanced

[38]CSRF stands for Cross-Site Request Forgery.

filters are available[39] (some of the most used filters include full text search and inequality operations that fetch only entries from before a certain date, for example). Table 4.1 shows some examples of queries and pagination.

`domain.com/api/data/` returns all the data entries
`domain.com/api/data/dataId/` returns the data entry whose id is equal to *dataId*
`domain.com/api/data/?data_type=DataType` returns all the data entries whose field `data_type` is set tp *DataType*
`domain.com/api/data/?static__filename__contains=.type` returns all the data whose `filename` field inside the `static` group contains `.type` – an example of a nested query with full text search
`domain.com/api/data/?limit=20&offset=20` pagination example, the query returns 20 data entries, skipping the first 20, effectively showing the second page of results

Table 4.1: API query examples, first line is the *URL* and the second the explanation

Testing

A very important aspect of both ensuring an application works and keeps on working correctly is testing. In the first stages of development, the server and the service API were extensively covered with unit tests to ensure the business logic was implemented correctly. The API was tested by sending mock requests to it and ensuring that it responded correctly and that all the actions were correctly logged. Once we believed that the tests sufficiently cover all aspects of the API, *test driven development* (TDD) [2] became the preferred process for adding new features. With TDD the developer first writes a failing unit test and then provides the functionality that the test

[39]Tastypie and *django-tastypie-mongoengine* support most of the filters available in Django's ORM system.

requires to pass. Eventually we planned to apply the same rigorous testing to the front-end apps as well, since this was one of the reasons that Angular was chosen.

4.2.3 Storage

As mentioned in the overview, we split storage into three parts – the relational part, the large document portion, and the file storage. The relational and document storage databases were kept loosely coupled from the server by using object-relational mapping (ORM, described in Section 2.1.5).

Relational storage

A portion of the data our platform needed to store is highly relational. Users, groups, and logs are interconnected with clear relations. Additionally, the Django server can use a relational database to manage sessions and allows access to the data through its administration interface. We chose PostgreSQL[40] as the relational database because it is a wide-spread, open source, free-for-commercial-use solution; it also has excellent integration with Django.

The actual objects that are stored in the database are defined at the server level and kept solution-independent with middle-ware engines that connect Python applications with databases and migration assisting packages which aid in propagating changes to object definitions[41]. Django provides an ORM solution which defines objects as *models* and then keeps them persistent in the chosen database.

Document storage

Above we explained that the platform defines data as an annotated piece of information. If the actual data are large files, they are saved in the file

[40] `www.postgresql.org`

[41] In PostgreSQL's case the middle-ware engine is *psycopg2* and migration support is provided by *south*.

system and their annotations are stored separately; if the text files are small enough they can be stored directly in the document storage. Because such a database has the potential to become very large in size, we considered NoSQL solutions. A survey of such solutions resulted in the choice of *MongoDB*[42], the most popular non-relational solution at the time. In the first stages of development, the size of the project definitely did not warrant the usage of a non-relational database, but MongoDB – a document storage database – proved to be a very suitable solution for our needs.

The interface of the platform works with JavaScript objects, defined in the *JavaScript Object Notation* (JSON). They are retrieved from the server API in that form and the transformation between JSON and Python objects is very simple (as long as some minor discrepancies are respected[43]). MongoDB stores the data in BSON, a binary form of JSON, resulting in the annotation being in more or less the same form throughout the platform, from the storage to the apps. MongoDB also supports nested queries of the BSON objects which further adds to the usability.

The document storage database is where the data, project, process, template, and trigger information is stored. The document schemata are defined with an ORM system through Django and related middle-ware. Despite using object-relational mapping, however, the final solution is highly customised for a specific database and would take much more work than just switching the middle-ware and storage solution to change to a different database, especially because the variety between different NoSQL databases is much higher than between relational ones.

The relational part of the database is referenced with IDs. For example, the documents store the primary keys of the users when they require a reference to them. Similarly, when a log stored in the relational database references a certain document it uses its (primary) key.

[42] `www.mongodb.org`

[43] Python tuples and numeric keys in dictionaries do not have equivalents in JavaScript and end up converted to lists and string keys.

File system storage

Some data entries are represented by either a large file or multiple (large) files. Such files are saved in a distributed file system inside folders corresponding to data IDs. What is stored depends on the type of data, when a new type is determined, the contents of the data folders of the type are defined as well. It is up to the processors accepting a certain data type as an input to correctly access the contents of the data folder, especially if the folder contains multiple files.

Persistence

There are two important issues that are very related to the way the storage system works – temporary and duplicate data. Every step of processing currently creates new data objects, which can be redundant if the user is only interested in the final result of a long chain of processes. At the same time, some identical processes can be run on the same data by multiple users, creating the same results – a waste of both computational power and storage space when the data is large and/or takes a long time to process.

A persistence system was planned to solve these issues but it was not fully implemented in the first few iterations of the solution. The idea was to use three levels of persistence: *raw* data is permanently stored, *cached* data is a copy of an existing raw data entry, and *temp* data represents temporary data that was used in a processing work-flow/pipeline but does not need to be stored.

Temporary data can easily be handled by setting the relevant parameters of processors that produce temporary data and scheduled garbage collection. The issue of duplicate data is a bit more complicated. First, a *checksum* function is needed for processors and their input parameters. With it, duplicate data that was created with the same version of the processor and the same parameters can be detected. Before beginning processing, the processing manager would need to check if the checksum matches any existing data entry. If it exists, the relevant portions of the original's annotation would be

copied in a new data entry, marked as cached, and pointed to the files of the original. Because the data cannot be changed after it is created, the only issue that needs handling is the potential deletion of the original raw data when it has cached copies pointing to it.

4.2.4 Processing

The processing portion of the application is managed by *Celery*[44], an asynchronous task queue based on distributed message passing. Celery uses MongoDB to store the messages and manage the queue. Tasks are put in the queue, checked that all their inputs are available, and executed on a pool of worker nodes inside Linux Containers (LXC). LXC execution ensures that a process is in a separate *control group* and isolated from the resources of the other processes, thus preventing it from affecting other parts of the system. The system saves progress updates that processors issue while they run and also ensures that all the data is correctly saved if processing finishes successfully, as well as ensuring that information is available in case of any errors. Portions of the data entries are written when the processor begins and ends the work and cannot be modified after the manager finishes the processing.

4.3 Selected implementation details

The following sections will take a closer look at some select key elements of the application.

4.3.1 Permissions

The platform uses an authorisation system that was inspired by the ones used in Google's products. There are five different rights a user can have over an object from the database: *view*, *edit*, *share*, *download*, and *add*. The view permission allows users read-only access to objects while edit allows them to

[44] `www.celeryproject.org`

modify and save the modifications to objects. The share right enables a user the ability to grant permissions to other users, the download right allows the user to access raw data and transfer it to their own computer. Finally, the add permission allows the user to add new data to a project.

All requests sent to the server API go through an authorisation step where the permissions of the user requesting a call are checked. Permissions are stored per-object in a JSON object where the user IDs are the keys for lists of permissions that were granted to those users. Users are not the only ones who have permissions, however, as the same logic applies to groups – this means that for every request we need to check whether a user has permission or if he is a member of a group which has the corresponding permission.

When a new data object is created, the permissions are propagated to it from the project the data object belongs to. Changes to permissions can be requested through the API by users with the share right and if permission changes are made to a project, the changes are propagated to the data objects belonging to the project as well – these changes supersede any changes to the data object. If a user creates a trigger, or tries to run a process manually, the authorisation system checks whether they have the required permission to add new data to the project.

There are two exceptions to the system described above. First is the super-user, a user at such a level bypasses all authorisation checks and is allowed to perform all operations. The other exception is the public user. The idea is that the platform should be accessible even to users who have not created accounts. For such users, an anonymous public user is automatically logged in because the whole service relies on sessions and a user being present. Public users have limited permissions; they are never allowed to share objects or add new data.

4.3.2 Triggers

The conceptual idea of triggers was explained in Section 2.2.4 and the eventual implementation was also touched upon. The key is allowing users to

select which portions of their data get automatically processed. Another role triggers serve is in running batch operations – automatic execution is optional. A trigger can be manually run, applying a processor to a selected subset of data.

A trigger is closely connected with a processor; this is the task it will be executing. One of the data inputs of this processor is replaced with a filter, which searches for matches inside the data annotation's schemata and template. If the filter is blank then all the data is accepted. The rest of the inputs are set to fixed values. Whenever the status of a data object changes, it checks if there are any triggers associated with it and if the conditions of the trigger are fulfilled. If both are true, the trigger will queue up a new task with the updated data object and the rest of the saved fixed inputs as the input data of the process.

Another idea that was explored, but not yet implemented, was how to solve a question of relevancy. A filter has its inputs fixed and has processed some relevant data. If we change the filter's inputs, the processed data no longer corresponds to the new settings of the filter. Currently, the only solution is to re-run the trigger, but that creates new data objects because the information about how the data was created cannot be changed. A proposed solution to this conundrum was adding a so-called *dirty bit*, a flag marking objects that are no longer up to date. Changing the settings of a filter would flag all the data that was created with it and then further propagate the dirty bit to any data that was created using the dirty data. The trigger settings would include an option to automatically re-run any dirtied data, or just leave it flagged and thus notify the user. The approach of re-running the dirtied data is incompatible with the initial idea of data being immutable after it is created.

4.3.3 Schemata and templates

The schemata and templates are an important portion of the application's structure because they are used to automatically generate the input and

display forms when data objects are viewed and created. This functionality is heavily used in the management app, but it can be used in other apps as well by either re-using the relevant Angular directives or by coming up with new ways to use the schemata and templates. A data object contains three schemata – the *static*, *input*, and *output* – and a template named *var* (for variable). Two schemata are defined by the processor, representing its inputs and outputs; the static one is the same for all data entries and represents the basic information, such as the name of the object, description, and the tags which apply to it. The definition of a template can also come from the processor in the form of a default template, or it can be defined on the project level. Default templates set at the project level ensure that all the data inside the project is annotated with the same information, although the template can still be changed on a per entry basis.

The goal of schemata and templates is to enable automatic generation of forms for the data and to handle the varied types of information that annotations can contain in a general manner. Because of that, the information contained inside a schema or template is similar to the way an input would be defined, including, but not limited to, whether the input is required, the placeholder text, default value, validation regular expression, even a list of possible values.

Another important piece of information is the type of the input – a dedicated type system was developed to differentiate between different inputs. There are two general groups of types, *basic* and *data*. The basic types are similar to those often found in programming languages, some examples are: `basic:boolean`, `basic:string`, `basic:date`, `basic:decimal`, `basic:json`. The data types represent outputs of processors and have `data:` prepended, these represent data entries in the database, for example `data:type:detail`. Additionally, lists of either *basic* or *data* types can be used, for example `list:basic:date` is the type for a list of dates.

All the supported basic types can be rendered in the interface with corresponding inputs, depending on the type and all the additional parameters,

for example, a list of possible values will result in a combo-box. A *data* type is rendered as a data table with an appropriate type filter set, from which a data entry of a valid type can be selected. Listing 4.2 shows an example of a simple schema and Figure 4.3 shows how that example schema can be rendered in the front-end application.

```
{ "schema": [
    {
        "default": "Default name",
        "label": "Name",
        "name": "name",
        "type": "basic:string"
    },
    {
        "default": [
            "tag1",
            "tag2"
        ],
        "label": "Tags",
        "name": "tags",
        "placeholder": "new tag",
        "type": "list:basic:string"
    },
    {
        "default": "Default description",
        "label": "Description",
        "name": "description",
        "type": "basic:text"
    }
]}
```

Listing 4.2: A schema example with three objects: name, tags, and description. Name and description use basic data types while tags are a list of basic strings. Figure 4.3 shows how this schema can be rendered in the interface.

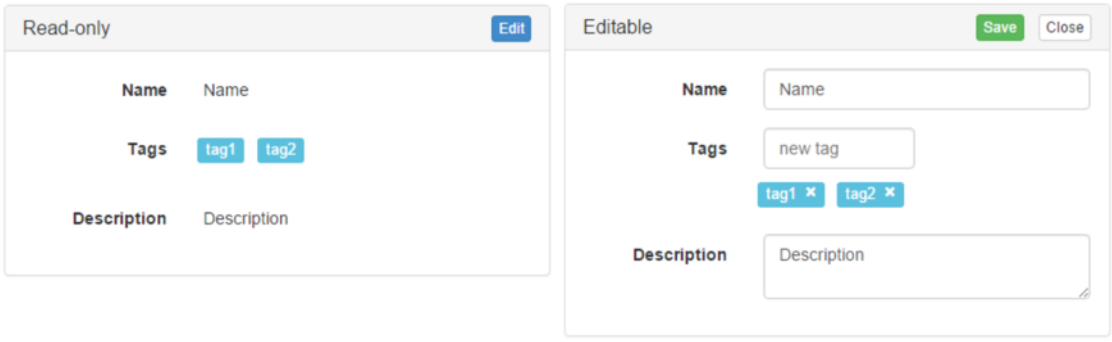


Figure 4.3: Schema 4.2 is rendered in read-only and editable mode with an angular directive.

Validation

The schemata and templates system is very useful but it needs to adhere to a common format to work. To ensure everything follows this format, a validation system based on JSON schemata[45] is used. A JSON schema defines the structure of the object and a validator can check if the object complies with the schema. In our case, we are using JSON-schemata (named meta-schemata) to validate the form of our schemata, templates, data types, and processors before they are saved into databases. Listing 4.3 shows an example of a JSON schema and valid JSON object corresponding to that schema.

4.3.4 Processors

A processor is an algorithm that accepts inputs and transforms them into outputs – it is the basic building block of the dataflow platform. Its definition is divided into four parts: the *inputs*, the *outputs*, the *meta-data*, and the *algorithm*. The inputs and outputs are schemata and follow all the rules that apply to them – their definitions, specifically on the input side, take into account how the end user will be selecting their values (for example

[45] `json-schema.org`

```
{
    "title": "Example Schema",
    "type": "object",
    "properties": {
      "firstName": {
        "type": "string"
      },
      "lastName": {
        "type": "string"
      },
      "age": {
        "description": "Age in years",
        "type": "integer",
        "minimum": 0
      }
    },
    "required": ["firstName", "lastName"]
}
{
    "firstName": "John",
    "lastName": "Doe",
    "age": 27
}
```

Listing 4.3: Example of a JSON schema and a valid JSON object according to the schema. First and last name are defined as strings and are required while the age is optional but has to be a natural number.

with combo or check boxes). The option to set default values for inputs is important as well. The meta-data of a processor is the information related to it and not the data entries it creates, including the version, name, description, and the persistence level of the created data. Finally, there is the algorithm that defines the process applied to the input data. It is written in *bash* and able to use any commands that are installed and available in the run-time environment. Essentially any code can be written, but the execution is limited by the (limited) permissions given to the worker executing the

algorithm.

Processors are JSON objects but we write them in the *YAML* (*YAML Ain't Markup Language*, previously *Yet Another Markup Language*)[46] markup. Parsing between the two is trivial but the YAML syntax is easier to write and clearer than JSON. Because processors are JSON objects we also apply the JSON schema-based validation to them, ensuring they are defined correctly and comply with the guidelines set by the developers. After writing, a processor needs to be registered with the platform through the command line. The register command validates the whole processor definition and adds it to the database. Listing 4.4 is a simple example of a processor definition that sleeps for a number of seconds as set by its input parameter.

4.3.5 Apps

At the beginning of this Chapter, we briefly described that apps are the interface through which the platform is accessed. The platform registers them and adds their static files to where they can be served. The app is defined with partial Django templates that the platform includes into the main framework. These partial templates hold the information needed to run a web application – the JavaScript libraries (*js.html*), style sheets (*css.html*), and the application itself (*content.html*). These templates end up embedded in the base website of the platform with a common header and footer, and the application renders in the content portion of the web page. A bare bones *hello world* example is a simple empty Django module with the *content.html* partial template defined, Figure 4.4 shows an example. More advanced apps will define their own processors and multiple sub-pages, but ultimately they all function in the same way – using the platform's API to process data and then request that data and display it in some meaningful way.

The first app running on the platform was the one that has been alluded to throughout this Thesis, the data and project management app named *GenCloud*. We expect that most users will be using it for project man-

[46] `yaml.org`

```
- name: test:sleep
  version: 1.0.0
  label: Test
  type: data:test:result
  persistence: TEMP
  description: Simple test running sleep for [t] seconds
  input:
    - name: t
      label: Sleep time
      type: basic:integer
      default: 5
  output:
    - name: output
      label: Result
      type: basic:string
  static:
    - name: name
      label: Name
      type: basic:string
      default: "Test"
    - name: tags
      label: Tags
      type: list:basic:string
      default: ["test"]
  var:
    - name: notes
      label: Notes
      type: basic:text
  run:
    runtime: polyglot
    bash: |
      echo "Starting..."
      sleep {{ t }}
      echo "{\"output\":\"/api/data/{{ data_id }}/download/
          stdout.txt\"}"
      exit 0
```

Listing 4.4: A simple YAML definition of a processor that sleeps for t seconds before exiting. The first portion of the definition is the meta-data of the processor, followed by the schemata (input, output, static) and the template (var), ending with *run*, which contains the algorithm. When the bash code from *run* is executed, the result is captured and stored by the processing manager.

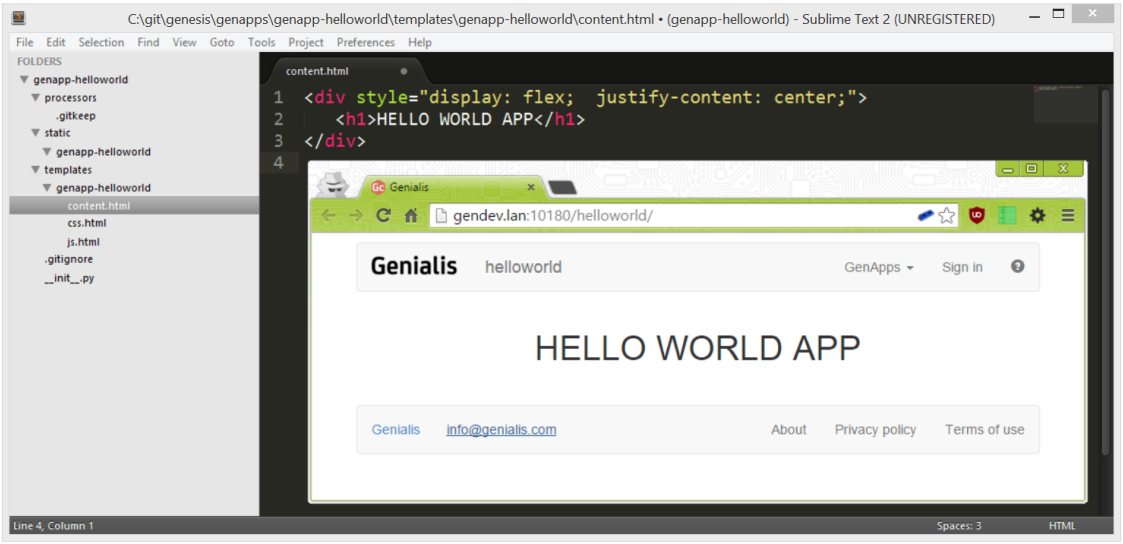


Figure 4.4: A minimal working *hello world* example app which can be added to the platform. The *css.html* and *js.html* partial templates contain the included css styles and JavaScript code, respectively, while the *content.html* contains the content of the app, in this case the hello world text.

agement as well as processing, but more advanced displays of data will be handled by more specialised apps. An example of such an app is the updated *dictyExpress*, shown in the Conclusion 5.2.

GenCloud

GenCloud is, first and foremost, a management tool through which projects and data can be administered. It is very general in scope, able to run any processors and set up triggers for them. The app keeps track of all the data inside a project, grouping them by type, and it also displays all the data that is currently being processed. The displays can be altered to also show already completed processes, Figure 4.5 shows how a simple project is displayed. The app handles triggers, both their creation, management, and their manual running. With the project settings the project permissions, default templates, and available processors can be modified, Figure 4.6 shows

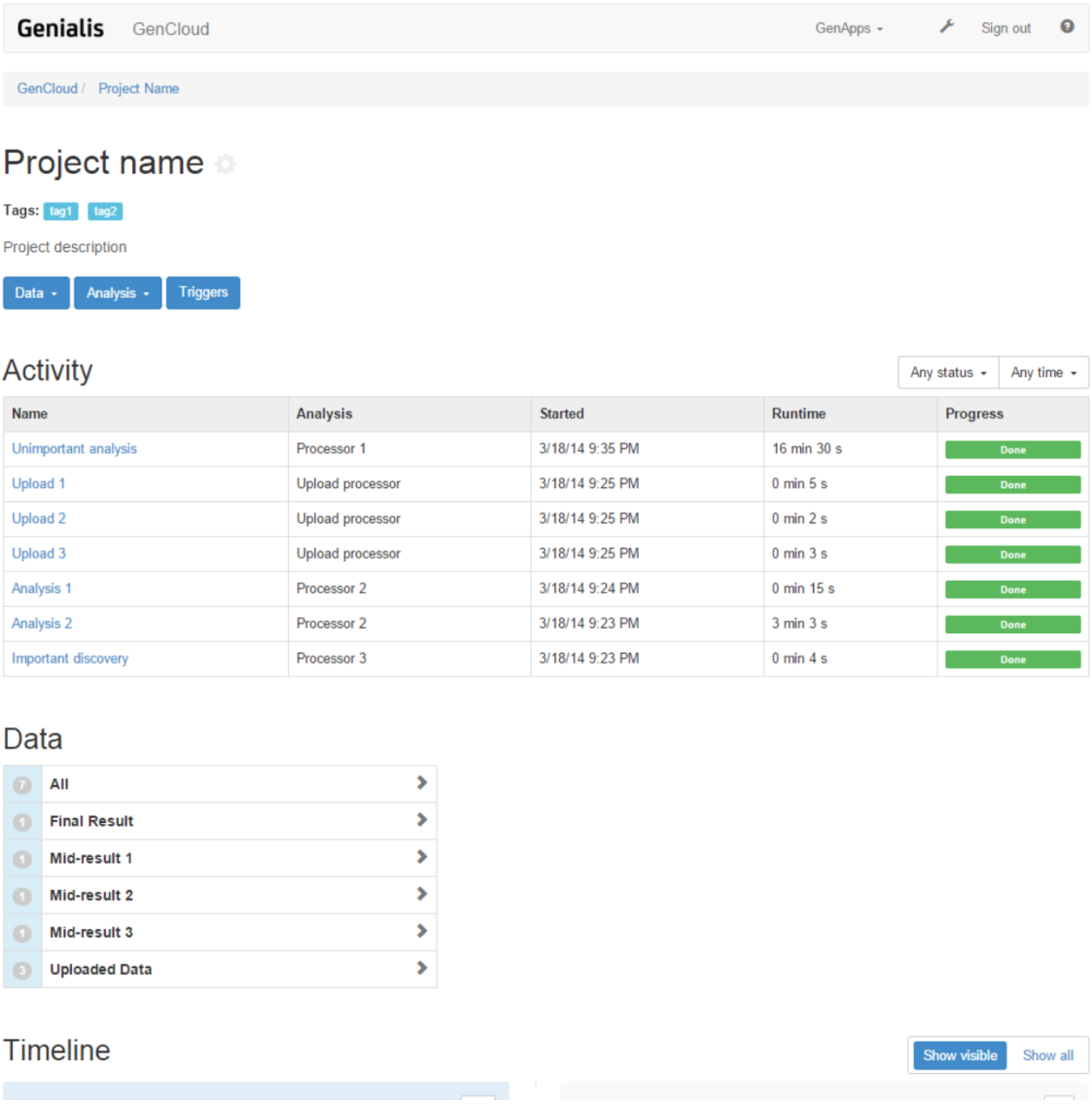


Figure 4.5: An example project within the application, showing the already completed analyses and how the data is grouped by type.

an example of this.

Each data entry can also be examined individually and here the system of schemata and templates comes into play – with an Angular directive named *dynamic*, the display of data is dynamically generated from a schema/template and the corresponding values. The created forms can either be read-

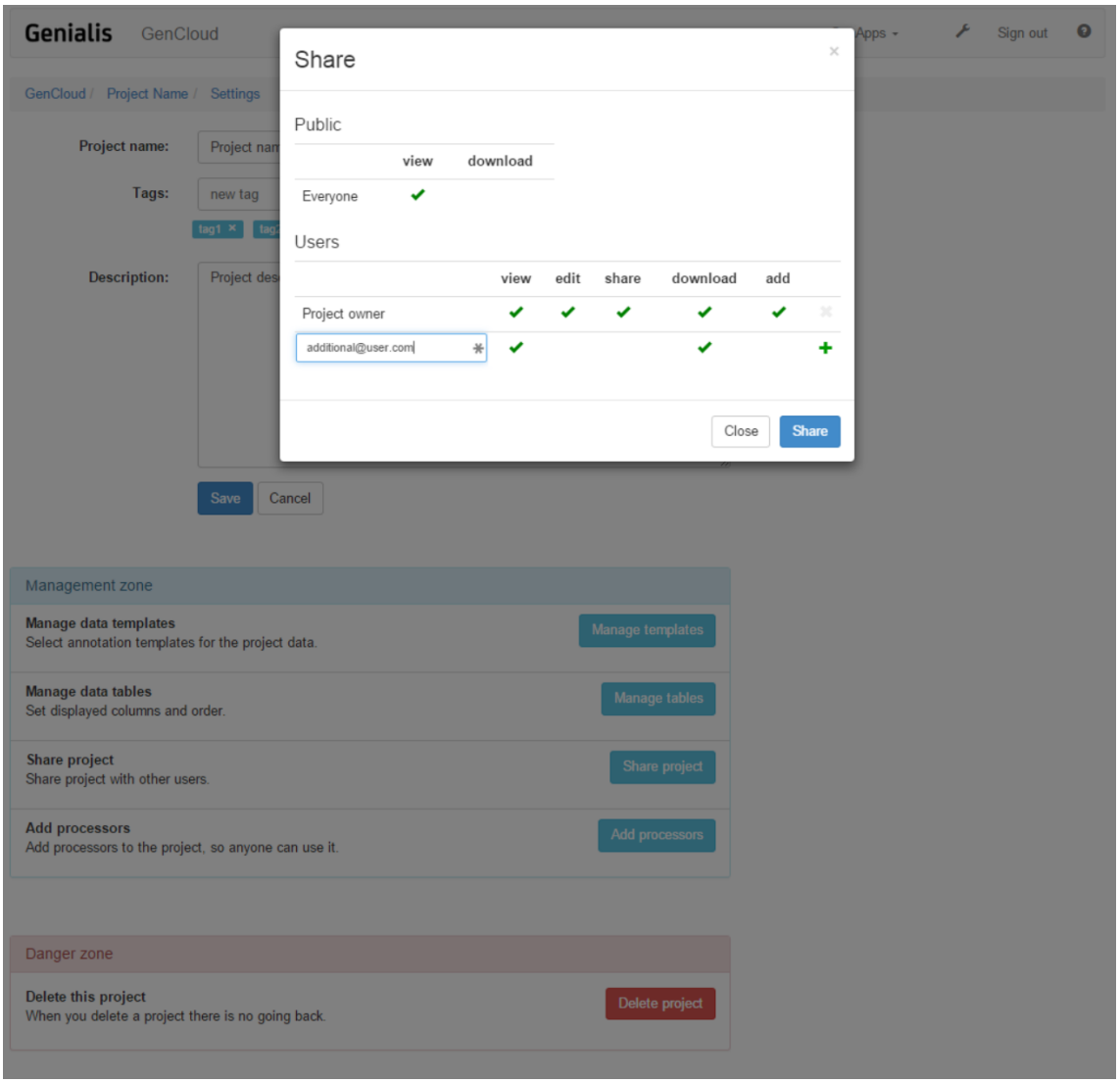


Figure 4.6: The foreground of the screenshot shows how the project permissions can be changed while the background shows the rest of the project settings that can be adjusted.

only or editable, the latter is used when data is being created or modified. This system is completely modular and the *dynamic* directive was developed for re-use in other apps that display data entries. Figure 4.7 shows an example of data details.

GenCloud was also imagined with a social aspect in mind – the project

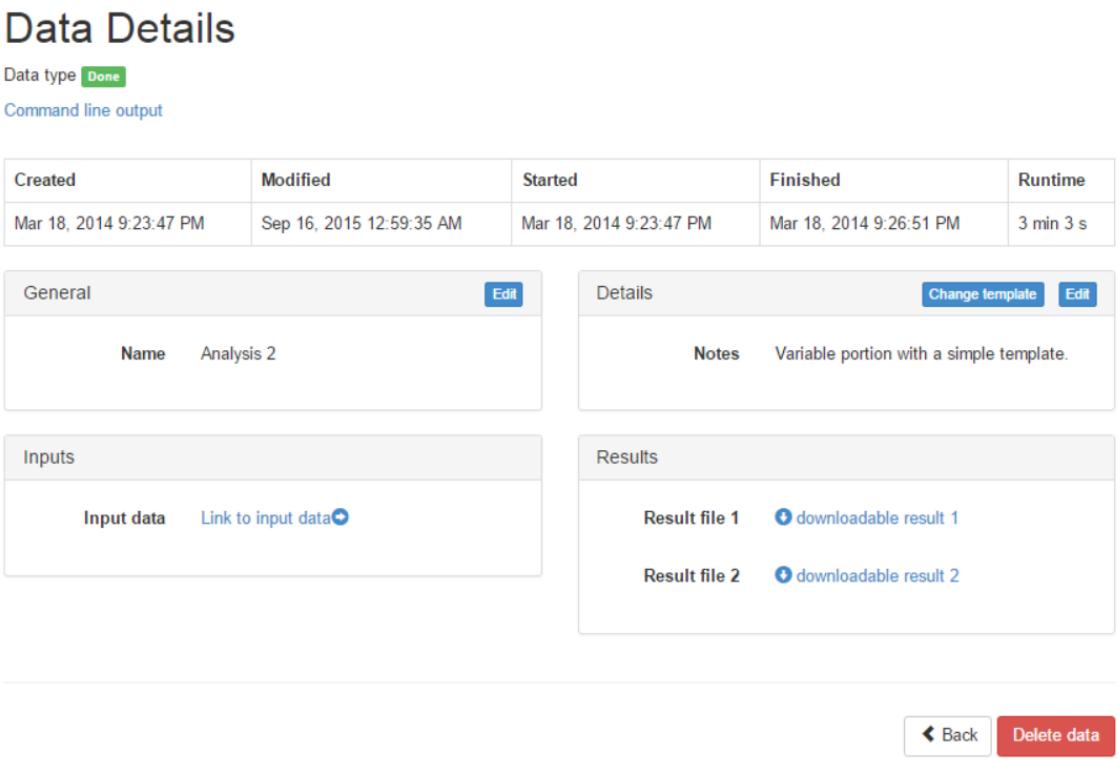


Figure 4.7: The annotation of a data entry is displayed in four sections – the general, inputs, and results correspond to the static, input, and output schemata, respectively, while the details portion represents the changeable template.

contains a timeline of events that are added whenever a processor is run, thus creating a history of all the processes that led to the current state of the project. Each of these events can be commented on or hidden when they are deemed not important enough, leaving the milestones visible. Int his way, the users of the platform could keep the comments and results of analyses and processes in a clearly visible timeline with key events highlighted. A new person, with whom the project was shared, would only have to check the timeline to see how the current state of the project came about.

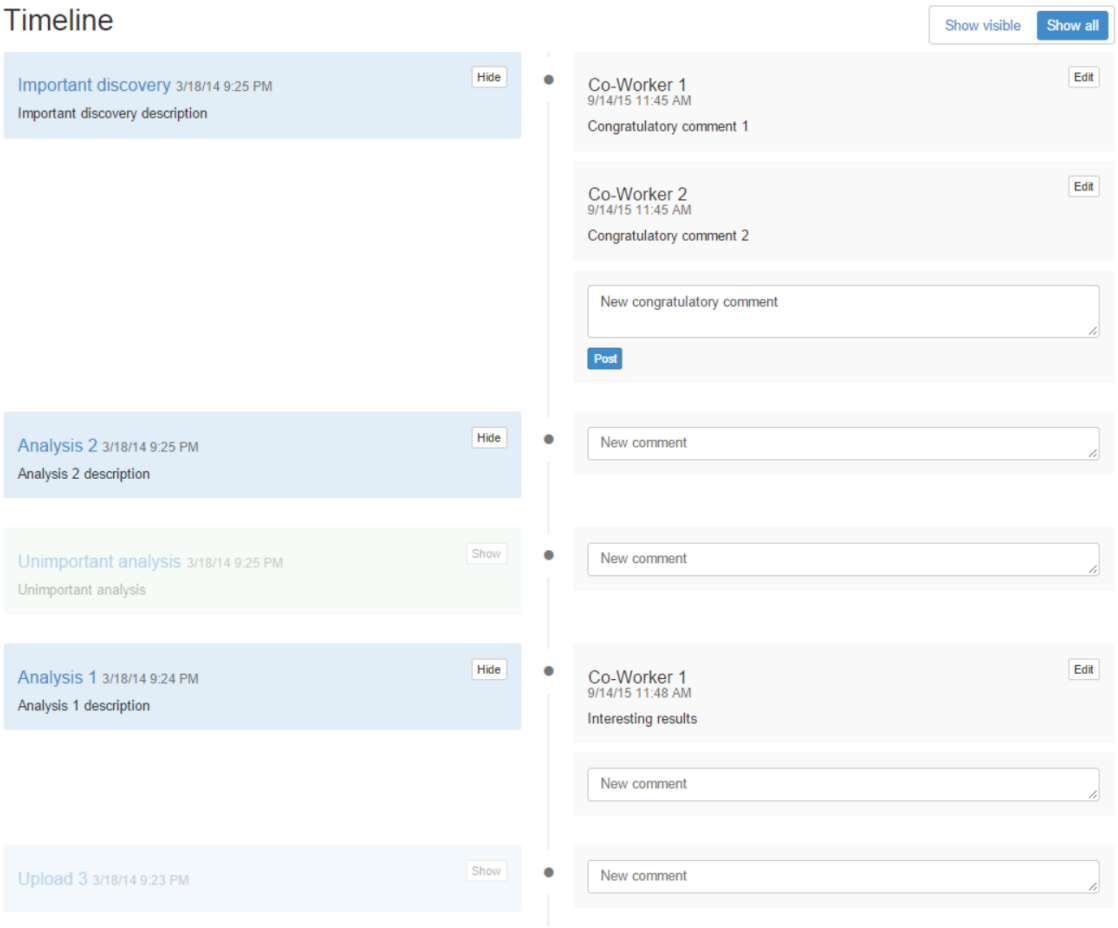


Figure 4.8: Project timeline, where users can comment on individual analyses and show/hide the ones they consider more/less important. The default view displays just the events that were not hidden, allowing for a quick overview of the important parts of the project.

Chapter 5

Conclusion

This work has presented the development of a data processing platform from the basic architectural decisions to a working solution that implemented most of the original ideas, creating a sold basis for further development. The platform was presented in its entirety, but the parts that the author either developed, or helped develop, were presented in greater detail. The author contributed to the development of the architecture of the application and then developed the first version of the web application and the corresponding web server and storage. From the first iteration onwards, the author continued to work on these two parts, focusing primarily on the RESTful API and business logic, as well as triggers and permissions. Ultimately, the platform described in this Thesis is the result of collaboration and plenty of discussion on both ideas and how best to materialise them.

How were the development goals fulfilled? The platform is able to scale horizontally on both the server and the storage level. It is as modular as the involved solutions allowed; integration with MongoDB, for example, is relatively high and would take plenty of work to switch to a different solution, whereas the relational portion can be swapped easily. The platform is highly extensible, allowing for the addition of both new processors and apps, which was demonstrated by using simple examples. The final goal – a platform that is simple to use – will have to be thoroughly tested. All the parts of the user

interface were built with ease-of-use in mind, but the level of success will be judged by the users, not the developers.

The original aim of creating a dataflow platform has been mostly fulfilled, especially with the utilisation of triggers to facilitate dataflow pipelines and automatic processing. Yet there is a crucial part that was never implemented – the visual dataflow interface. Bound by time and man-hour constraints, the implementation of such an interface was not a priority. We do believe, however, that because all the major functions of the application can be easily controlled through the RESTful API, the platform is ready for such a visual interface, should someone at some point decide to develop it.

5.1 Looking back

The author concluded his involvement with the development of the platform in May 2014. Now, as we are concluding this Thesis, we can look back and reflect on the decisions that were made. The technological choices were mostly sound, for example, AngularJS did become the most prominent of the MV* frameworks used for web application development. MongoDB is still the most popular NoSQL database but the trend of using NoSQL databases for most web projects has cooled off because of the realisation that most projects never reach the volumes of data that would warrant such databases. Fresh trends are emerging – *NewSQL*[47] is a new generation of relational databases that promises a combination of proper transaction support through SQL as well as scalability. Nonetheless, MongoDB served us well because of the way we used it, resulting in data in the same format across the platform.

When the development began, on demand platforms for data analytics were not readily available. Since then some of the largest players have entered the game, releasing their own versions of on demand tools, essentially creat-

[47]Michael Stonebraker's article on the topic is a good introduction, available at `cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext`

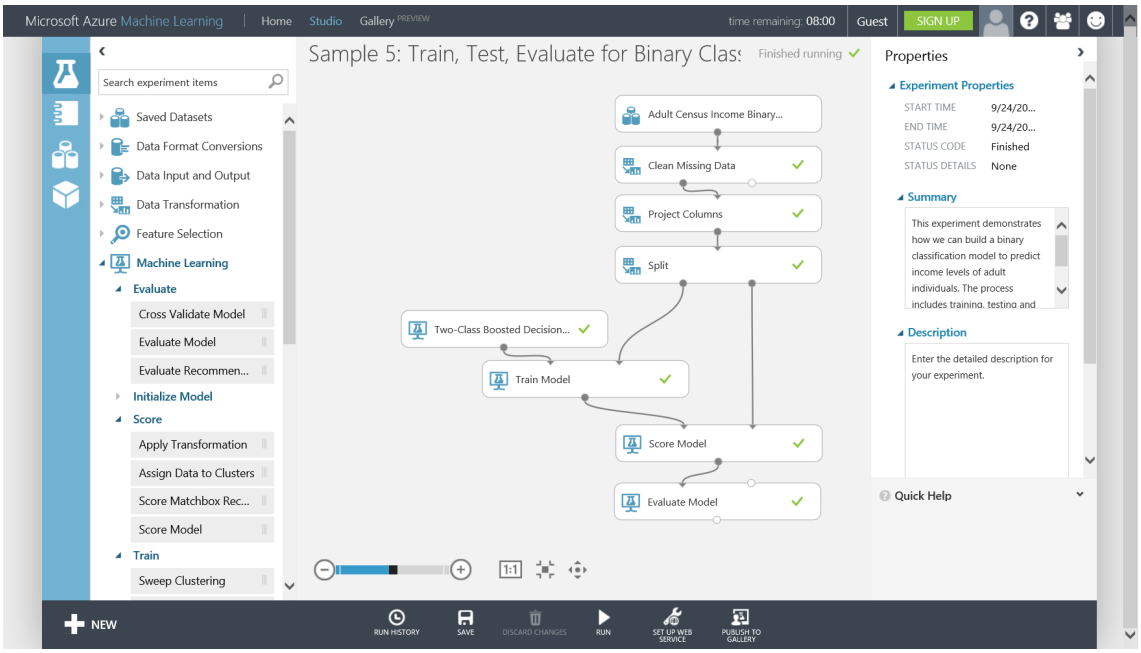


Figure 5.1: Microsoft Azure Machine Learning with a visual dataflow interface.

ing *Machine-Learning-as-a-Service*: *Microsoft Azure's Machine Learning*[48], *Amazon Machine Learning*[49], *Google Prediction API*[50], *IBM's Watson Analytics*[51], and more. All such solutions try to provide a robust and accurate service that is simple enough for anyone with data to use. Various means are chosen to achieve the goals, from opting for a simple API in Google's case to Microsoft's *Machine Learning Studio*, an elaborate visual dataflow interface (Figure 5.1). The market is rapidly developing and we look forward to seeing what will happen in a few years, especially in the case of solutions that do not have the backing of large multinational companies.

[48] `azure.microsoft.com/en-us/services/machine-learning`
[49] `aws.amazon.com/machine-learning`
[50] `cloud.google.com/prediction`
[51] `www.ibm.com/analytics/watson-analytics`

5.2 Looking forward

The platform is being actively developed and many parts of the original implementations have been upgraded, sometimes even replaced with better ones. The same development has occurred in the ideas, some endured while others (for example the timeline) were abandoned. The most exciting portion is the development of specialised apps that fully utilise what the platform can do, an example of such an app is *dictyExpress*.

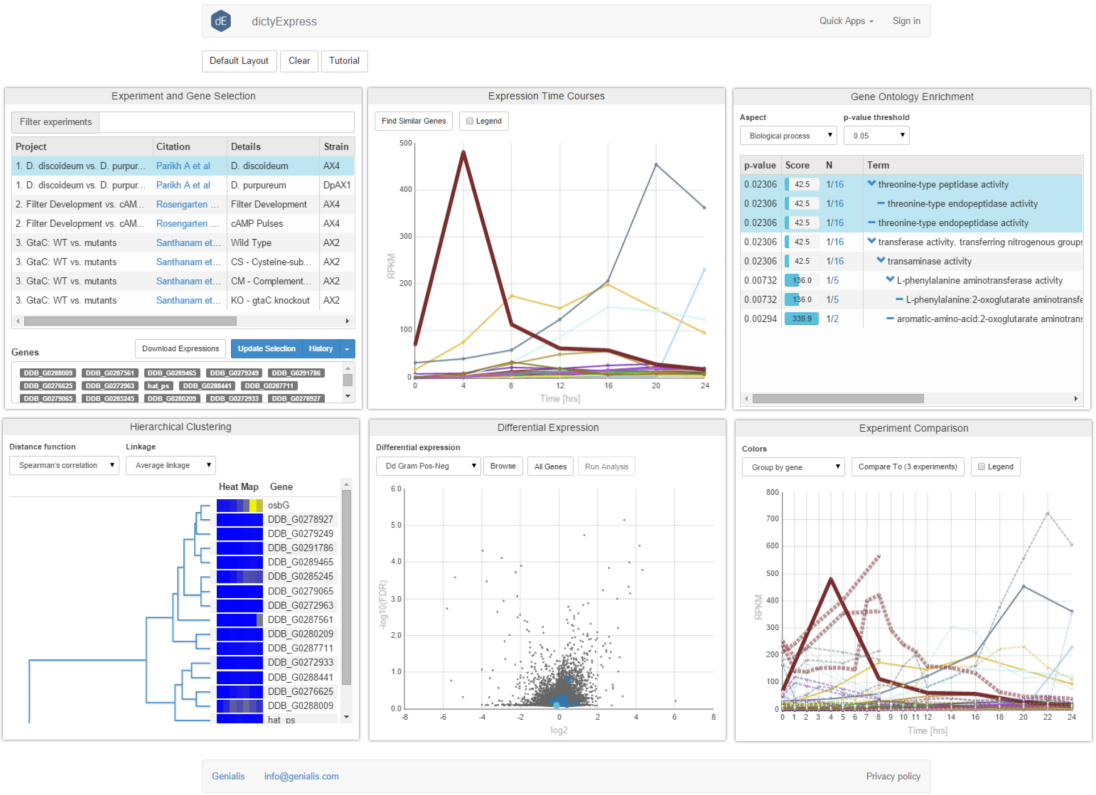


Figure 5.2: The interface of the updated dictyExpress application.

dictyExpress[52] was designed as an updated version of the application described in Section 3.2. Development was done in cooperation with the Bioinformatics Laboratory at the University of Ljubljana as well as Gad Shaulsky's and Adam Kuspa's labs at Baylor College of Medicine. It finished

[52] `dictyexpress.research.bcm.edu`

in early 2015. The function of the app is still the same – it is an interactive, exploratory data analytics tool that provides access to gene expression experiments in *Dictyostelium* – but the ideas were refined and updated to better suit the new technologies and meet the needs of its users. The app fully utilises the developed platform with custom processors and an extensive array of functionality that first processes data on the platform and then fetches it for the elaborate visualisations. Figure 5.2 provides an example of the updated interface.

The new dictyExpress is just one example of the kind of applications Genialis is now developing, in collaboration with the University of Ljubljana, to tackle the complex problems of data analysis in bioinformatics. We are pleased to report that the ideas presented in this Thesis took off and continue to be improved in a start-up environment.

References

[1] E. Baroth and C. Hartsough, "Visual Object-oriented Programming," M. M. Burnett, A. Goldberg, and T. G. Lewis, Eds. Greenwich, CT, USA: Manning Publications Co., 1995, ch. Visual Programming in the Real World, pp. 21–42.

[2] K. Beck, *Test Driven Development: By Example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel, "KNIME: The Konstanz Information Miner," in *Data Analysis, Machine Learning and Applications*, C. Preisach, H. Burkhardt, L. Schmidt-Thieme, and R. Decker, Eds. Springer Berlin Heidelberg, 2008, pp. 319–326.

[4] D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor, "Galaxy: A Web-Based Genome Analysis Tool for Experimentalists," *Current protocols in molecular biology*, pp. 19–10, 2010.

[5] A. B. Bondi, "Characteristics of Scalability and Their Impact on Performance," in *Proceedings of the 2nd International Workshop on Software and Performance.* New York, NY, USA: ACM, 2000, pp. 195–203.

[6] E. Brewer, "CAP twelve years later: How the"rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, Feb 2012.

[7] E. A. Brewer, "Towards Robust Distributed Systems (Abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2000, pp. 7–.

[8] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data Mining Toolbox in Python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013.

[9] E. W. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982, pp. 60–66.

[10] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[11] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002.

[12] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[13] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, W. C. Miller, W. J. Kent, and A. Nekrutenko, "Galaxy: a platform for interactive large-scale genome analysis," *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[14] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biol*, vol. 11, no. 8, p. R86, 2010.

[15] J. Gray, "The Transaction Concept: Virtues and Limitations (Invited Paper)," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. VLDB Endowment, 1981, pp. 144–154.

[16] T. Haerder and A. Reuter, "Principles of Transaction-oriented Database Recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983.

[17] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in Dataflow Programming Languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, Mar. 2004.

[18] S. Joosten, "Trigger Modelling for Workflow Analysis," in *Proceedings of the Ninth Austrian-informatics Conference on Workflow Management: Challenges, Paradigms and Products: Challenges, Paradigms and Products*. Munich, Germany: R. Oldenbourg Verlag GmbH, 1994, pp. 236–247.

[19] R. N. Katz, P. J. Goldstein, and R. Yanosky, "Demystifying Cloud Computing for Higher Education: Highlights of Cloud Computing," *Research Bulletin, Issue 19*, 2009.

[20] G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80," *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988.

[21] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski, "Scale-up x Scale-out: A Case Study using Nutch/Lucene," in *IPDPS 2007. IEEE International*, March 2007, pp. 1–8.

[22] J. P. Morrison, *Flow-Based Programming, 2nd Edition: A New Approach to Applictaion Development*. Paramount, CA: CreateSpace, 2010.

[23] ——, "Flow-Based Programming," *Journal for Developers of Heterogeneous Computing Systems*, vol. 1, no. 1, 2013.

[24] G. Papadopoulos and K. Traub, "Multithreading: a revisionist view of dataflow architectures," in *Computer Architecture, 1991. The 18th Annual International Symposium on*, 1991, pp. 342–351.

[25] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[26] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision," in *Proceedings of the 17th International Conference on World Wide Web*. New York, NY, USA: ACM, 2008, pp. 805–814.

[27] D. Pritchett, "BASE: An Acid Alternative," *Queue*, vol. 6, no. 3, pp. 48–55, May 2008.

[28] G. Rot, A. Parikh, T. Curk, A. Kuspa, G. Shaulsky, and B. Zupan, "dictyExpress: a Dictyostelium discoideum gene expression database with an explorative data analysis web-based interface." *BMC Bioinformatics*, p. 265, 2010.

[29] C. Russell, "Bridging the object-relational divide," *Queue*, vol. 6, no. 3, pp. 18–28, May 2008.

[30] M. E. Skinner, A. V. Uzilov, L. D. Stein, C. J. Mungall, and I. H. Holmes, "JBrowse: A next-generation genome browser," *Genome Research*, vol. 19, no. 9, p. 1630–1638, Jul 2009.

[31] C. Snijders, U. Matzat, and U.-D. Reips, "Big Data": Big Gaps of Knowledge in the Field of Internet Science." *International Journal of Internet Science*, vol. 7, no. 1, 2012.

[32] T. B. Sousa, "Dataflow Programming: Concept, Languages and Applications," 2012, available at `paginas.fe.up.pt/~prodei/dsie12/papers/paper_17.pdf`.

[33] W. Stevens, G. Myers, and L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[34] W. M. P. van der Aalst, "The Application of Petri Nets to Workflow Management." *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.