Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO

Departamento de Computación

"Olote: plataforma para la ejecución de experimentos in silico en un cluster de cómputo"

Tesis que presenta

Víctor Manuel Villa Moreno

Para Obtener el Grado de

Maestro en Ciencias

En la Especialidad de

Computación

Directores de la Tesis:

Dr. Sergio Víctor Chapa Vergara

Dr. Mauricio Carrillo Tripp

México, Distrito Federal. **Noviembre, 2013**

Resumen

Presentamos el diseño e implementación de una plataforma Web que simplifica el proceso de ejecución de experimentos in silico en el cluster de cómputo del Laboratorio Nacional de Genómica para la Biodiversidad. Con esto se busca mejorar la experiencia de los usuarios para así aprovechar de mejor manera los recursos de cómputo de alto rendimiento del laboratorio. Los programas que serán ejecutados en el cluster se definen de una manera sencilla mediante objetos YAML, a partir de los cuales los usuarios podrán definir experimentos mediante formularios HTML y diagramas de flujo construidos con gráficos SVG. Se utilizó una arquitectura orientada a servicios, la cual ofrece parte de la funcionalidad de la plataforma a través de una interfaz REST. El resultado de este trabajo es la reducción del tiempo y complejidad relacionados con la ejecución de experimentos in silico para los investigadores del laboratorio.

Abstract

We present the design and implementation of a Web platform aimed at simplifying the execution of in silico experiments in the computer cluster of the National Laboratory of Genomics for Biodiversity. Our aim is to improve the users experience and consequently to improve the use of high performance computing resources in this institution. The programs to be run on the cluster are easily described through YAML objects and from these, users can define experiments using HTML forms and workflows drawn with Scalable Vector Graphics. A service oriented architecture was used, allowing part of the platform's functionality to be reused through a REST application programming interface. A reduction in the time and complexity required for the execution of in silico experiments in this laboratory was achieved as a result of this work.

Agradecimientos

La realización de este trabajo no hubiera sido posible sin el apoyo de mi familia, la dirección de los Doctores Mauricio y Sergio, así como el patrocinio del Consejo Nacional de Ciencia y Tecnología

Índice general

Índice de figuras

Índice de códigos

Capítulo 1

Introducción

El estudio de macromoléculas biológicas se ha convertido en parte esencial del trabajo de investigadores de diversas áreas de la biología. Esto se debe a que dichas moléculas son las protagonistas principales del proceso conocido como el *dogma central de la biología molecular*, que ayuda a explicar muchas de las propiedades de los organismos.

Este estudio es una tarea compleja para la cual es indispensable el uso de herramientas de cómputo, lo cual ha dado lugar al nacimiento de la bioinformática como disciplina. Sin embargo, el uso y combinación de estas herramientas no es siempre sencillo. El software desarrollado para esta tesis busca simplificar y automatizar esta tarea para los estudiantes e investigadores del Laboratorio Nacional de Genómica para la Biodiversidad (Langebio), parte del Cinvestav Irapuato.

1.1. Problema a abordar

Aunque en la bioinformática existen programas interactivos y con una interfaz gráfica, tales como visualizadores de modelos en tres dimensiones, una parte importante de los programas usados en el área son diseñados para procesamiento por lotes. Este tipo de programa tiene las siguientes características:

- se ejecuta en una consola de texto mediante un comando que define opciones de ejecución y apunta a archivos de entrada;
- en muchos casos se desarrollan solamente para sistemas operativos basados en Unix;
- la ejecución se llevará a cabo sin interacción por parte del usuario y su duración dependerá de la complejidad de la tarea;
- cuando la ejecución termine el resultado habrá sido impreso en la terminal o guardado en uno ó más archivos.

Un análisis bioinformático no trivial involucra trabajar con archivos de datos muy grandes y una demanda alta de recursos de cómputo como procesador y memoria,

que rápidamente superan las capacidades de las computadoras personales. Por esta razón es común que los análisis de este tipo se realicen en clusters de cómputo de alto rendimiento. El uso de un cluster de cómputo típicamente involucra los siguientes pasos adicionales por parte del usuario para la ejecución de un programa:

- conectarse al cluster mediante un protocolo como *secure shell connection* (ssh);
- crear un script con el comando a ejecutar y enviarlo al manejador de colas para generar un *job* en una cola;
- monitorear el estado de cada job en la terminal;
- transferir los archivos de entrada y resultados, hacia y desde el cluster.

En muchos casos los investigadores no son necesariamente usuarios expertos en el manejo de clusters, por lo que se encuentran con una curva de aprendizaje larga. Además, este flujo de trabajo requiere mucha preparación manual, lo que dificulta su automatización por el usuario.

En el Langebio se cuenta con un cluster de cómputo de alto rendimiento nombrado Mazorka, que cuenta con los programas de análisis bioinformático más utilizados por los investigadores de la institución, tales como *blast*, *PhyML*, *Trinity*, entre otros. Mazorka es un cluster de cómputo heterogéneo: contiene conjuntos de nodos con distintas características, como cantidad de memoria y velocidad del procesador, agrupados lógicamente como colas de ejecución. Este diseño se debe a que los requerimientos de recursos computacionales son distintos para distintos tipos de análisis bioinformáticos e idealmente un job será ejecutado en la cola de ejecución más adecuada para el tipo de análisis. Sin embargo, esto trae consigo un problema adicional, ya que es común que los usuarios elijan una cola de ejecución inadecuada.

1.1.1. Proceso actual

A continuación se listan de manera específica los pasos que un investigador en el Langebio típicamente sigue para realizar un análisis en Mazorka.

1. Elaborar un script similar al del Código 1.1.
2. Enviar a Mazorka este script y cualquier archivo de entrada necesario para el análisis mediante un comando como `scp`.
3. Conectarse al cluster mediante `ssh` y enviar el script al administrador de colas con el comando `qsub ejemplo.sh` para generar un *job*, obteniendo así un identificador de job.
4. Monitorear constantemente el estatus del *job* en el cluster con el comando `qstat`, refiriéndolo mediante el identificador de job.

5. Una vez finalizado el job, localizar los archivos de salida en la terminal y, de ser necesario, transferirlos a la computadora del usuario nuevamente utilizando el comando `scp`.

Código 1.1: Script para ejecutar un programa en Mazorka

```
#PBS -N blastx
#PBS -l nodes=1:ppn=8
#PBS -q twin
#PBS -V

cd $PBS_O_WORKDIR

blastall -p blastx -d /home/secuencias/nr -v 5 -b 5 -a 8 \
-i 454Isotigs.fna -o contigs.nr.txt
```

1.1.2. Workflows

En los análisis bioinformáticos —también llamados experimentos in silico— es común utilizar la salida de un programa como entrada de otro. A este *encadenamiento* de análisis lo denominaremos un *workflow*. La ejecución de workflows es común en experimentos in silico. Sin embargo, dado el flujo de trabajo mencionado en la sección anterior, la ejecución de un workflow implica la ejecución manual de un análisis después de otro y no es fácil incorporar mecanismos para automatizar su ejecución.

1.2. Propuesta

En este trabajo proponemos el desarrollo de una herramienta computacional que simplifique el diseño de experimentos in silico para los investigadores del Langebio y su ejecución en Mazorka. Para esto, se diseñará e implementará una plataforma Web que permitirá a sus usuarios la ejecución de experimentos en un cluster de cómputo mediante una interfaz que constará de formularios Web y un editor visual para el diseño de workflows.

1.2.1. Motivación

La idea original fue implementar un sistema similar al desarrollado por el Instituto Max-Planck [1], que permite a los investigadores la interacción con las herramientas computacionales mediante una interfaz Web y además permite usar los resultados generados por un programa, una vez terminada su ejecución, como entrada a otro programa, proporcionando así la capacidad de crear workflows de manera semi automática.

1.2.2. Objetivos generales

La plataforma se diseñará e implementará de tal manera que tenga las siguientes características.

No disruptiva

Aunque se busca cambiar la manera en que los usuarios diseñan y ejecutan experimentos, la plataforma deberá ser simplemente un intermediario entre el usuario y el cluster que simplifique el uso del último. Los archivos de entrada de los análisis y los archivos de salida que generan no deberán cambiar y el usuario deberá de tener el mismo acceso a ellos que tenía antes.

Simple

La administración de la plataforma incluye acciones como la adición de nuevos programas. Además, se espera que los usuarios del Langebio puedan modificar la plataforma para satisfacer sus necesidades futuras. Por estas razones se tiene como objetivo seleccionar lenguajes, técnicas y herramientas con la consideración de que en el futuro la plataforma podrá ser modificada por investigadores y estudiantes de bioinformática cuyo tiempo y recursos para aprender estas tecnologías son limitados.

Reutilizable

La funcionalidad de ejecución de experimentos de la plataforma puede ser de utilidad aún cuando no se utilice su interfaz Web. Esta funcionalidad deberá estar accesible para otros programas.

Extensible

La bioinformática es una disciplina joven en la que nuevos programas siguen escribiéndose y nuevas técnicas desarrollándose. Es por esto que la plataforma debe permitir la incorporación de nuevos programas de análisis y el uso de nuevos formatos de archivos de datos.

1.2.3. Objetivos particulares

Para alcanzar los siguientes descritos en la sección anterior se realizarán las siguientes acciones.

- Definir un modelo de datos que abstraiga a los programas que serán ejecutados en el cluster.
- Definir el modelo de datos de un job, que contenga la información necesaria para la ejecución de uno de estos programas.

- Definir un modelo de datos de un workflow, que represente un conjunto de jobs y las dependencias entre ellos.
- Desarrollar un conjunto de servicios Web que produzcan y procesen estos modelos de datos para su ejecución en el cluster.
- Desarrollar una aplicación cliente que consuma estos servicios para mostrarle al usuario una interfaz Web en la que pueda ejecutar experimentos mediante formularios Web y diagramas de flujo.

1.2.4. Hipótesis

El uso de una herramienta visual para el diseño y ejecución de experimentos in silico en un cluster de cómputo, que tenga como principio fundamental la simplicidad en su uso y configuración, contribuye al mejor uso de los recursos de cómputo de alto rendimiento por parte de los investigadores de áreas biológicas.

Capítulo 2

Marco teórico y estado del arte

En este capítulo abordaremos brevemente varios conceptos de los diferentes temas relacionados a este trabajo. Comenzaremos con la bioinformática, que es el área de estudio para la cuál se empleará la plataforma desarrollada en este trabajo. Después explicaremos algunas de las tecnologías y conceptos utilizados en el diseño y desarrollo de esta plataforma. Finalmente, comentaremos sobre las herramientas disponibles actualmente cuyo que proporcionan una solución similar a nuestra propuesta.

2.1. Bioinformática

La bioinformática es el área interdisciplinaria que surgió a partir de la necesidad de almacenar, consultar, organizar y analizar información biológica. La constante generación de datos ha llevado a que muchos de los retos en biología sean ahora retos para la computación. La bioinformática es la aplicación en gran escala de técnicas computacionales para analizar la información asociada con biomoléculas. Se ha establecido firmemente como una disciplina dentro de la biología molecular y comprende un amplio rango de áreas, desde biología estructural y genómica hasta el estudio de la expresión de genes [2]. Hoy en día, es una parte importante del trabajo de muchos científicos de las áreas biológicas.

2.1.1. Bases biológicas

Francis Crick describió en 1958 [3] el flujo de información genética en un organismo, bautizándolo como el *dogma central de la biología molecular*. Las tres moléculas principales de participan en este flujo son el *ácido desoxirribonucleico* (ADN), *ácido ribonucleico* (ARN) y *proteínas*. El proceso principal de este flujo es la generación de proteínas a partir del ARN, que se genera, a su vez, a partir del ADN. Este proceso influye en el estudio de prácticamente cualquier área de la biología, porque es así como el ADN, transferido de una generación a otra, codifica las instrucciones genéticas utilizadas en el desarrollo y funcionamiento de todos los organismos vivos conocidos.

Las proteínas, el ADN y el ARN son polímeros lineales, es decir, cadenas moléculas formadas por la unión de monómeros donde cada monómero forma como máximo dos enlaces. El ADN esta formado por una cadena doble de nucleótidos, donde cada nucleótido contiene una de cuatro posibles bases nitrogenadas diferentes: *adenina* (A), *citosina* (C), *guanina* (G) o *timina* (T). Mediante un proceso conocido como transcripción, secuencias de ARN son generadas a partir de segmentos de ADN. El ARN esta formado por una cadena sencilla de los mismos nucleótidos que el ADN, con la excepción de que las bases nitrogenadas de timina son reemplazadas por *uracina*. A partir del ARN se generan proteínas mediante un proceso conocido como traducción, donde diferentes combinaciones de tres bases nitrogenadas generan un solo aminoácido.

2.1.2. Consideraciones computacionales

La información genética que estas tres moléculas contienen esta dada por su secuencia: el orden de sus elementos codifica información. Una molécula hipotética de DNA con la secuencia AAACCCGGGTTT, después de ser transcrita y traducida, producirá una cadena de aminoácidos distinta a una molécula con la secuencia GGGTTTAAACCC, aunque contengan exactamente el mismo tipo y número de bases.

La principal consideración al trabajar con estas moléculas es su tamaño. El genoma más pequeño conocido esta formado por casi $160,000$ elementos, mientras que el genoma humano esta formado por alrededor de 3.2×10^9 pares de bases nitrogenadas. Además de esto, el ADN de cada especie de organismo es distinto y es posible estudiar organismos extintos obteniendo su ADN de restos fósiles o comparando organismos vivos relacionados para encontrar el ADN de antepasados comunes, por lo que existe una cantidad inmensa de información biológica que se puede estudiar. GenBank [4], una de las bases de datos de secuencias genómicas más usadas, en su versión liberada en febrero de 2013 contiene más de 162 millones de secuencias y su tamaño crece de manera exponencial, duplicándose su número de secuencias cada 18 meses.

Por estas razones el uso de herramientas computacionales se ha vuelto fundamental en la biología. La demanda de recursos computacionales en muchos de estos casos es tan grande, que la única manera práctica de ejecutar algunos análisis es mediante clusters o grids de computadoras.

Análisis in silico de datos

En biología existen distintos términos para describir como se realiza un experimento, como *in vivo*, *in vitro* e *in situ* para describir experimentos realizados en organismos vivos, fuera del organismo y en el lugar donde se encuentren estos organismos en la naturaleza, respectivamente. El término *in silico* surgió como analogía a estos para describir un análisis realizado en computadora o experimento realizado mediante una simulación por computadora.

2.2. Tecnologías empleadas en el desarrollo de la plataforma

Para entender mejor las herramientas que forman el estado del arte actual y posteriormente el diseño e implementación de la plataforma, primero abordaremos las principales tecnologías involucradas.

2.2.1. Clusters de cómputo de alto rendimiento

Un cluster de cómputo de alto rendimiento es un sistema de cómputo paralelo formado por un conjunto de nodos independientes, donde cada nodo es un sistema individual capaz de operar independientemente y armado con componentes tradicionales [5]. En la actualidad son populares en el ámbito científico los clusters denominados de tipo *Beowulf*, construidos a partir de nodos que utilizan hardware que no fue desarrollado específicamente para cómputo paralelo o de alto rendimiento y un sistema operativo libre como Linux o BSD. Las ventajas de este tipo de cluster son su bajo costo, flexibilidad y accesibilidad [5].

Administradores de ejecución

La ejecución de programas en un cluster típicamente se realiza mediante un administrador de ejecución de procesos, que es un sistema que controla la ejecución programas en la misma computadora o en un sistema remoto. Proporcionan un único punto de control para la definición y monitoreo de procesos. En el caso de un cluster, un administrador de ejecución puede agrupar los nodos del cluster en conjuntos lógicos llamados colas de ejecución. Uno de los administradores de ejecución más populares en los clusters de cómputo científicos y el utilizado en el LANGEBIO es el *Portable Batch System*.

2.2.2. Patrones de arquitectura de software

Conforme incrementan el tamaño y complejidad de los sistemas de software, los problemas de diseño van más allá de los algoritmos y estructuras de datos utilizados: diseñar y especificar la estructura global del sistema emerge como un nuevo problema. Los problemas a solucionar en este nivel son las estructuras de control y organización global, protocolos de comunicación, sincronización y acceso de datos, entre otros [6]. Al diseño en este nivel de abstracción se le conoce como arquitectura de software y a continuación mencionamos algunas técnicas de arquitectura empleadas en esta tesis.

Patrón Modelo-Vista-Controlador

El patrón de arquitectura de software conocido como *Modelo-Vista-Controlador* tiene como idea principal la separación lógica entre las interfaces de usuario, los datos y la lógica de un programa [7]. Bajo este estilo de diseño las estructuras de datos, clases

y lógica relacionada con la representación de los datos se agrupa lógicamente como el *modelo*; la parte del programa encargada de la presentación de una interfaz y datos al usuario se agrupa como la *vista*; y, finalmente, a la lógica que se encarga de realizar la conexión entre estos dos grupos se le conoce como el *controlador*.

Al desarrollar aplicaciones interactivas la separación de sus componentes tiene enormes beneficios. Aislar cada unidad funcional tanto como sea posible facilita al diseñador de la aplicación cada unidad sin tener que preocuparse por los detalles de las otras unidades. Por ejemplo, si se desea reemplazar la interfaz de una aplicación o proporcionar interfaces en distintos idiomas, será un proceso más sencillo en una aplicación diseñada con un patrón modelo-vista-controlador [8].

Arquitecturas orientadas a servicios

Un servicio es la abstracción de cierta funcionalidad de un sistema con una interfaz definida, de manera que otros componentes del sistema pueden utilizar o consumir esta funcionalidad. Una arquitectura orientada a servicios implica diseñar una aplicación de manera que partes de su funcionalidad se ofrezcan como servicios y estos sean consumidos por otros componentes de la aplicación. Así como otras maneras de aislar la lógica en un sistema, esta separación permite reemplazar o modificar servicios, así como explotar la funcionalidad ofrecida por los servicios por nuevos componentes o incluso otras aplicaciones [9].

Representional State Transfer

El *Representional State Transfer*, o *REST*, es un estilo de arquitectura de software que toma sus principios del funcionamiento del protocolo *HTTP* descrito por Roy Fielding, uno de los principales creadores de HTTP y arquitecto de la Web. Sus fundamentos más característicos son:

- Sigue una arquitectura cliente-servidor
- Sus transacciones no tienen estado
- Presenta interfaces uniformes Además de ser uniformes, las interfaces deben ajustarse a las siguientes restricciones:
  - Identificación de recursos
  - Manipulación de recursos mediante representaciones
  - Mensajes auto-descriptivos
  - Hipermedia como el motor del estado de la aplicación

Que una transacción no tenga estado (llamadas *stateless* en inglés) significa que son independientes una de otra, de manera que cada transacción contiene la información necesaria para comprender la solicitud y no puede utilizar ningún tipo de

contexto almacenado en el servidor. De esta manera el estado de la sesión se almacena completamente en el cliente. Este método presenta tres principales ventajas: visibilidad, confiabilidad y escalabilidad. La visibilidad mejora porque para monitorear al sistema es suficiente analizar una transacción de manera independiente para comprender su naturaleza. La confiabilidad mejora porque durante un fallo en el sistema se reducen las transacciones parciales. La escalabilidad mejora porque el servidor puede liberar recursos inmediatamente después de una transacción y múltiples transacciones pueden incluso realizarse en distintos servidores definidos mediante un balanceo de cargas.

La característica central de REST es el énfasis en una interfaz uniforma entre componentes. Cualquier información a la que se le puede dar un nombre puede ser un recurso: un documento, una imagen, un servicio, una colección de otros recursos, etc. En otras palabras, cualquier concepto que pueda ser el objetivo de una referencia de hipertexto debe pasar la definición de un recurso. Un recurso es un mapa conceptual a un conjunto de entidades, no la entidad que corresponde al mapeo de ese recurso en un punto particular de tiempo.

2.2.3. Lenguajes visuales

Un lenguaje visual es un lenguaje que permite transmitir un mensaje mediante elementos visuales. Un lenguaje de programación visual es un sistema de símbolos, que puede o no incluir elementos textuales, que permite describir los pasos computacionales requeridos para efectuar una tarea mediante representaciones en dos dimensiones. Esto en contraste con los lenguajes tradicionales de programación, que son textuales y se les considera como de una dimensión [10].

El uso de lenguajes de programación visual ha sido explorado en varias áreas tales como en la enseñanza de la programación, la creación de contenido multimedia, entre otros [11]. En particular, se ha trabajado con ellos dentro del Cinvestav en la consulta de bases de datos [12] [13]. En [14] se explora el uso de un lenguaje de programación visual, denominado *Modelo Meta-conceptual Dinámico* para la consulta de bases de datos biológicas. En este trabajo se explora el uso de un lenguaje visual para la creación de flujos de trabajo para el análisis de datos in-silico.

2.2.4. Tecnologías Web

La plataforma desarrollada en este trabajo funciona sobre tecnologías Web. Desde la manera en que está disponible a los usuarios hasta como está estructurada internamente alrededor del concepto de recursos.

Servicios web

Un servicio Web es un servicio, como se describió en la sección sobre arquitectura, que es accesible a través de una red para proveer interacción máquina-máquina. Tiene

una interfaz descrita en un lenguaje procesable por una máquina y típicamente otros sistemas interactúan con el servicio mediante HTTP.

Universal Resource Identifier

Para identificar recursos a lo ancho de una red de dispositivos de cómputo interconectados se utiliza un *universal resource identifier* o *URI*, que es una cadena de texto que identifica de manera única a un recurso dentro de esa red. Tal identificador permite la interacción con representaciones de un recurso en la Web a través de una red.

Hypertext Transfer Protocol

El *Hypertext Transfer Protocol* (HTTP) es un protocolo a nivel de aplicación usado en la comunicación de sistemas distribuidos y relacionados entre si mediante hipermedia. Es un protocolo de tipo cliente-servidor y solicitud-respuesta, en el que mediante un URI un cliente envía una solicitud a un servidor y este envía de vuelta una respuesta. Existen varios *métodos* (también llamados *verbos*) que definen el tipo de solicitud, siendo los más comúnmente usados *GET*, *POST*, *PATCH* y *DELETE*.

Hypertext Markup Language

Desarrollado a finales de 1990, el *HyperText Markup Language* es un lenguaje de marcado —un lenguaje que incorpora a un documento etiquetas para describir su contenido— originalmente pensado para compartir documentos electrónicos que permitieran incluir apuntadores a otros documentos. Hoy en día este lenguaje es la base sobre la que diseñan todas las páginas Web.

HTML se utiliza para describir el contenido de un documento, encerrando entre un par de etiquetas los distintos elementos de una página. El conjunto de etiquetas válidas está definido en el estándar HTML [15] elaborado por el *World Wide Web Consortium (W3C)*. Las etiquetas se colocan entre los símbolos '<' y '>'. La mayoría de las etiquetas requiere una etiqueta de cierre, idéntica a la de apertura pero precedida por el símbolo '/'. Por ejemplo, el título de una página Web con noticias lo especificamos con: *<title>Noticias</title>*.

El estándar HTML 4.01 se finalizó en 1999 y por más de diez años el W3C no publicó un nuevo estándar. Durante este tiempo se desarrollaron nuevas tecnologías y los fabricantes de navegadores Web agregaron sus propias extensiones no estándar a la especificación HTML. La proliferación de estas extensiones provocó problemas de interoperabilidad entre navegadores. El W3C, mientras tanto, concentraba sus esfuerzos en una nueva versión de *XHTML* que nunca vio la luz. Al mismo tiempo, un grupo ajeno al W3C, el *Web Hypertext Application Technology Working Group (WHATWG)*, desarrollaba una especificación para una nueva versión de HTML. En esta versión, conocida como HTML5 [16], se optó por estandarizar algunas de estas extensiones, de manera que la funcionalidad que ya se tenía en los navegadores fuera más claramente definida y así adoptada de manera oficial por el resto de los

fabricantes. Este esfuerzo fue posteriormente adoptado por el W3C y se considera actualmente el estándar oficial, aunque su estatus es de borrador.

Javascript Object Notation

El *Javascript Object Notation (JSON)* [17] es un formato para intercambio de datos que tiene las características de ser basado en texto, ligero e independiente de un lenguaje. Define un conjunto corto de reglas de formato para representar datos estructurados de manera portátil. Se deriva de la notación utilizada para objetos literales en Javascript. Este formato se utilizara en el proyecto para almacenar la información sobre los programas y los flujos de trabajo.

Un texto en JSON esta compuesto por una secuencia de tokens. Los tokens validos incluyen seis caracteres estructurales, cadenas, números, y tres literales. Los seis caracteres estructurales son: el de inicio de un arreglo '[', el final de un arreglo ']', el inicio de un objeto '', el final de un objeto '', el separador de nombre y valor ':' y el separador de elementos ','. Las literales válidas son *true*, *false* y *null*.

JSON puede representar cuatro tipos primitivos: cadenas, números, booleanos y nulo; y dos estructuras: objetos y arreglos. Una cadena es una secuencia de cero o más caracteres. Un número, al igual que en la mayoría de los lenguajes de programación, esta formado por un componente entero, prefijado por un signo negativo opcional, que puede estar seguido de una parte decimal o exponencial. Un objeto es un conjunto de cero o más pares de nombre y valor, donde un nombre es una cadena y un valor puede ser una cadena, número, booleano, null, un arreglo u otro objeto. Un arreglo es una secuencia ordenada de valores.

En el Código 2.1 incluimos la definición de algunos de los tipos de datos de JSON utilizando el *Augmented Backus-Naur Form (ABNF)* descrito en [18].

Código 2.1: Gramáticas para la representación de algunos tipos de datos en JSON

```
# Definicion de un numero
  number = [ minus ] int [ frac ] [ exp ]
  decimal-point =  .
  digit1-9 =  1-9
  e =  e E
  exp = e [ minus / plus ] 1*DIGIT
  frac = decimal-point 1*DIGIT
  int = zero / ( digit1-9 *DIGIT )
  minus =  -
  plus =  +
  zero =  0


# Definicion de un grupo
  object = begin-object [ member *( value-separator member ) ]
        end-object
  begin-object = {
  end-object = }
```

```
# Definicion de un arreglo
  array = begin-array [ value *( value-separator value ) ]
          end-array
  begin-array = [
  end-array = [
```

En el Código 2.2 mostramos el ejemplo de un objeto JSON que contiene varios de sus tipos de datos. Comienza por un grupo llamado *Cazadores de microbios*, cuyo contenido es un arreglo de objetos representando personas. Estos objetos incluyen a su vez otros objetos que guardan información relacionada a la persona, como su nombre, año de nacimiento y un arreglo que incluye cadenas con algunas de sus aportaciones a la ciencia.

Código 2.2: Ejemplo de un objeto JSON

```
{
  "Cazadores de microbios": [
    {
      "Nombre completo": "Antonie Philips van Leeuwenhoek",
      "Aportaciones": [
        "Mejoramiento del microscopio",
        "Creacion de la microbiologia"
      ],
      "Nacimiento": 1632
    },
    {
      "Nombre completo": "Louis Pasteur",
      "Aportaciones": [
        "Vacunacion",
        "Fermentacion Microbiana",
        "Pasteurizacion"
      ],
      "Nacimiento": 1822
    },
    {
      "Nombre completo": "Robert Heinrich Herman Koch",
      "Aportaciones": [
        "Fundador de la bacteriologia"
      ],
      "Nacimiento": 1843
    }
  ]
}
```

2.2.5. YAML

YAML es un lenguaje para serialización de datos legible y sencillo de escribir que es compatible con JSON, por lo que lo utilizamos en varios lugares de la plataforma. En el Código 2.3 mostramos un objeto YAML equivalente al del Código 2.2 en JSON.

Código 2.3: Ejemplo equivalente en YAML

```
Cazadores de microbios:
  - Nombre completo: Antonie Philips van Leeuwenhoek
    Nacimiento: 1632
    Aportaciones:
     - Mejoramiento del microscopio
     - Creacion de la microbiologia

  - Nombre completo: Louis Pasteur
    Nacimiento: 1822
    Aportaciones:
     - Vacunacion
     - Fermentacion Microbiana
     - Pasteurizacion

  - Nombre completo: Robert Heinrich Herman Koch
    Nacimiento: 1843
    Aportaciones:
     - Fundador de la bacteriologia
```

2.2.6. Expresiones Regulares

Una expresión regular (también conocida como *regex* del inglés *regular expression*) es una manera formal de describir secuencias de caracteres dentro de una cadena.

2.3. Estado del arte

El siguiente es un listado de las herramientas computacionales disponibles que, a nuestro conocimiento, cumplen funciones similares a las propuestas en este trabajo.

2.3.1. Diseño y ejecución visual de workflows científicos

Existen varias herramientas que permiten a un usuario diseñar un workflow científico, sin embargo en ninguna de estas la ejecución se realiza directamente en un cluster de cómputo.

Taverna Workflow Management System

Taverna [19], creado en la Universidad de Manchester, es un programa de escritorio desarrollado en Java diseñado en un inicio para ejecutar workflows biológicos que después fue adaptado para trabajar con workflows científicos en general. Esta diseñado para trabajar con servicios Web, principalmente los ofrecidos en BioCatalogue [20] [21], sin embargo es posible añadir servicios manualmente especificando su URI [22].

Cuenta con numerosas características además del diseño visual de workflows, como la ejecución de workflows desde la línea de comando, ejecución remota de workflows en un servidor de Taverna, búsqueda de servicios Web integrada, entre otros. También posible definir un programa local como un servicio [23].

Aun con estas capacidades, no existe una manera directa de ejecutar los workflows en un cluster local y no en los servicios Web proveídos en los catálogos. Como alternativa para esto se desarrolló la extensión TavernaPBS.

TavernaPBS

Esta es una extensión [24] para Taverna que permite diseñar workflows que serán ejecutados en un cluster mediante un administrador de colas PBS. La manera en que TavernaPBS funciona es esencialmente extendiendo el ambiente conocido como *beanshell* [25] dentro de Taverna. Beanshell es un lenguaje interpretado basado en Java y esto implica que para utilizar TavernaPBS para ejecutar workflows en un cluster se requiere escribir al menos un poco de código en Beanshell [26].

Kepler

Kepler [27] es un programa desarrollado en Java para el análisis y modelado de datos científicos mediante representaciones visuales de estos procesos. Los componentes de un workflow en Kepler son llamados actores y la instalación de la herramienta incluye más de 350 actores listos para usarse. Entre estos se incluyen actores en los lenguajes R y Matlab para el análisis estadístico de datos. Estos actores se ejecutan de manera local, pero también se incluye un actor llamado WebService para la ejecución remota de procesos dentro de un workflow local.

Galaxy

Galaxy [28] es un proyecto desarrollado en Python para la ejecución de workflows científicos, integración de datos y publicación de análisis y datos. Fue diseñado por investigadores de las universidades de Penn State y Emory. A diferencia de Taverna y Kepler, Galaxy funciona como una aplicación Web. El uso recomendado de Galaxy es en el servidor central localizado en [29], pero puede descargarse para usar una versión local si se necesitan desarrollar extensiones, utilizar nuevas fuentes de datos o agregar nuevas herramientas [30].

A menos que se use una instalación local, las herramientas que usa Galaxy para construir workflows son servicios en línea. En el caso de que se agreguen herramientas localmente se debe crear un archivo XML [31] que describe el programa a ejecutar.

2.3.2. Generación automática de servicios Web

Como se explicará en el siguiente capítulo, una parte fundamental en el diseño de la plataforma que proponemos es la generación automática de servicios Web para la

ejecución de distintos programas en el cluster a partir de archivos de configuración. Es decir, no es necesario modificar un programa o generar código especial para cada programa que se agregue a la plataforma, sino que definimos una abstracción que representa un programa del cluster y generamos servicios Web solamente con un archivo de configuración.

GFac

El *Generic Service Toolkit* (GFac2) [32] es un proyecto desarrollado en la Universidad de Indiana para la ejecución de análisis y workflows científicos en un grid de cómputo distribuido. Es una solución completa y robusta desarrollada en Java que permite .[en]volver.[a]plicaciones de línea de comando y genera una interfaz HTML para su ejecución en un grid. Sin embargo esta es una solución a gran escala que involucra ejecuciones en sitios remotos donde la aplicación debe también ser instalada. Si se desea ejecutar programas a través de un administrador de colas como PBS se debe utilizar el sistema Globus [33] y su extensión GRAM [34].

Jersey

En Java existen propuestas como *Jersey* [35], que permite extender una clase para que proveerá una interfaz REST como un servicio Web. Esto requiere acceso y modificaciones al código fuente, además de estar limitado a código en Java.

La funcionalidad de generar servicios Web automáticamente para la ejecución remota de programas bioinformáticos enfocada a un cluster de cómputo de alto rendimiento a partir de simples archivos de configuración, hasta donde tengo conocimiento, no ha sido propuesta anteriormente.

Capítulo 3

Análisis

A continuación describiremos las ideas principales que definirán el funcionamiento de la plataforma y algunas de las decisiones que se tomaron cuando existen distintas tecnologías que cumplen propósitos similares.

3.1. Funcionamiento general

La plataforma funcionará, de manera general, como un intermediario entre los usuarios y Mazorka. Para esto, deberá contar con una interfaz gráfica con la cual puedan interactuar los usuarios al mismo tiempo que deberá contar con acceso al cluster para realizar la ejecución de programas y acceso a los archivos de resultados. Dadas estas características, existen dos opciones: que sea un programa que el usuario ejecute en su computadora y este se conecte al cluster; ó que la plataforma exista en el cluster y los usuarios se conecten a ella. Elegimos la segunda opción porque así la plataforma se encuentra centralizada en una única ubicación. Esto tiene las siguientes ventajas:

- La configuración de la plataforma se realiza en una sola ocasión para todos los usuarios. Por ejemplo, pueden agregarse o modificarse los programas que se deseen ejecutar en el cluster sin necesidad de distribuir parches o archivos de configuración.
- La seguridad de la plataforma se configura en un solo lugar.
- En caso de liberarse una nueva versión de la plataforma solo es necesario cambiarla en el cluster y los usuarios accederán a ella de manera automática.

Si la plataforma se ejecutará en el cluster entonces el usuario necesita una manera de interactuar con ella. Esto se realiza mediante una aplicación cliente, lo que le da a la plataforma una arquitectura *cliente-servidor*. Para simplificar el diseño se optó por implementar la plataforma como una aplicación Web. De esta manera, el usuario utilizará un navegador Web para interactuar con la plataforma.

3.2. Interacción con Mazorka

La manera en que un usuario ejecuta experimentos en Mazorka una vez que tiene acceso a una terminal virtual a través de `ssh` es enviando scripts de ejecución al programa `qsub`, consultando el estatus de un job con `qstat` y una vez que termine, analizando los archivos de salida. Si la plataforma se ejecuta en el cluster como se describió en la sección anterior, entonces lo único que necesita hacer es ejecutar los mismos comandos para comunicarse con el administrador de recursos del cluster y proporcionarle al usuario los archivos de salida.

Mazorka corre sobre un sistema operativo CentOS Linux y se creará un usuario para ejecutar la plataforma. Bajo este usuario será que Olote enviará trabajos al administrador de recursos.

3.3. Abstracción de recursos

Para la configuración de los programas que se desean ejecutar en el cluster, Olote utilizará un formato de serialización de información en archivos de texto. De esta manera los programas se pueden configurar directamente con un editor de texto y respaldar de una manera sencilla. En la sección 5 se explica su formato y contenido.

Por su versatilidad y amplio soporte entre lenguajes de programación para aplicaciones Web existen dos formatos principales para definir estos modelos de datos: XML y JSON. XML es definido por el *World Wide Web Consortium* y es una aplicación del *Standard Generalized Markup Language*. Para el desarrollo de Olote se eligió JSON por la simplicidad para desarrollarlo, leerlo y modificarlo, comparado con XML. Además, se eligió el lenguaje de serialización YAML, ya que produce objetos JSON de una manera aún más amigable.

3.4. Interacción con el cliente

Como medio de interacción entre el usuario y la plataforma para la definición de experimentos se decidió emplear las características inherentes de HTML. Olote utilizará formularios HTML como medio principal para permitir al usuario la definición de los parámetros de ejecución de los programas y la creación de diagramas de flujo para describir workflows mediante gráficos vectoriales, una nueva adición al estándar HTML5. De esta manera no es necesario desarrollas nuevas maneras de entrada de datos y se utilizan características ya presentes en la mayoría de los navegadores Web modernos.

3.5. Ofrecimiento de servicios

Para permitir el reuso de código e incrementar la flexibilidad de la plataforma se decidió ofrecer parte de su funcionalidad como servicios Web. La funcionalidad que

se ofrecerá es la relacionada a la ejecución de programas y workflows en el cluster, lo que permitirá que se pueda hacer esto utilizando un cliente mientras este siga los esquemas definidos para la plataforma. El formato elegido para los servicios Web es usando objetos JSON, en lugar de objetos XML, por las mismas razones que se eligió JSON para los archivos que definen los programas: son más sencillos de implementar y posteriormente de modificar.

Capítulo 4

Diseño y arquitectura

Basándonos en las decisiones discutidas en el capítulo anterior podemos ahora explicar los principios que usamos para el diseño de Olote. De manera general, la plataforma es una aplicación Web que funcionará como interfaz para la ejecución de experimentos in silico en un cluster de cómputo que utilice el administrador de recursos PBS. La interacción entre el usuario y la plataforma será mediante un navegador Web que le mostrará formularios HTML para especificar los parámetros de ejecución de los programas que se ejecutarán en el cluster. Además, le permitirá especificar un conjunto de programas con dependencias de resultados entre ellos para ejecutar un *workflows* de análisis. Su diseño gira alrededor de dos decisiones principales: la separación de su funcionalidad en capas para permitir que los servicios que ofrece sean consumidos por otras aplicaciones, y la abstracción de los programas a utilizar en la plataforma mediante una estructura de datos fácilmente representada en un archivo de texto legible y editable.

4.1. Separación en capas

La plataforma se dividió en dos capas, el *backend* y el *frontend*, no solo con la intención de agrupar funcionalidad, sino también con el propósito de ofrecer parte de esta funcionalidad como servicios que pudieran ser consumidos por otros programas.

En la figura 4.1 se representan las dos capas que forman la plataforma. También se representa el administrador de recursos *PBS* como una capa, con el fin de abstraer de manera más clara a los actores que participan en el proceso de ejecución de programas en el cluster.

4.1.1. Backend

En esta capa se agrupa la funcionalidad relacionada con:

- Lectura de los archivos de configuración de los programas y creación de representaciones de estos en memoria,

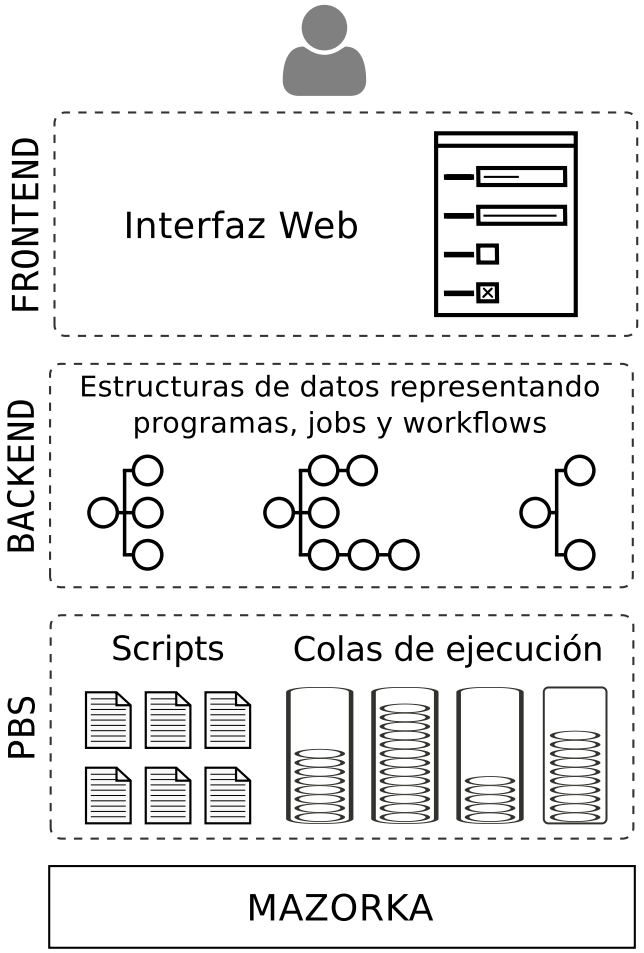

Figura 4.1: Arquitectura general de la plataforma

- Responder a peticiones que soliciten estas representaciones,
- Recibir y validar solicitudes de ejecución de estos programas,
- Generar los archivos necesarios para transformar esta solicitud en una solicitud procesable por PBS.

De esta manera, el backend es la capa que se encarga de generar y recibir representaciones de un programa —como se describe en la sección 5— y comunicarse con PBS para la ejecución de los programas y la consulta de su estado.

Toda la funcionalidad de esta capa se ofrece como servicios Web con JSON, de esta manera la funcionalidad de ejecutar programas y workflows mediante la plataforma se puede ofrecer sin la necesidad de utilizar el frontend Web de la plataforma. Los tres servicios principales, como se muestra en la figura 4.2, que ofrece esta capa son: *tools*, *jobs* y *workflows*.

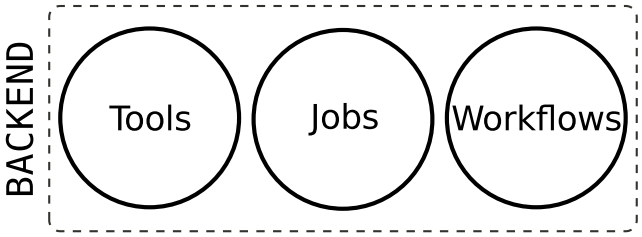

Figura 4.2: Servicios que ofrece la capa *backend*

Tools

Definimos como *tool*, o herramienta, a cada uno de los programas que podrán ser ejecutados a través de la plataforma. Sus características principales son las listadas en la sección 1.1 y la manera de definirlos en la plataforma se explica en la sección 5, pero por el momento es necesario mencionar que la manera de agregarlos y editarlos es mediante la edición de archivos de texto directamente en Mazorka. Es decir, no existen métodos que mediante HTTP realicen algún tipo de escritura en las herramientas.

Dado lo anterior y siguiendo las convenciones REST, los métodos que se pueden ejecutar sobre los recursos *tools* son: *index* y *show*. Ambas acciones provocan que se lean los archivos de texto de las herramientas. En el caso de la acción *show*, el *backend* devolverá una estructura JSON que contiene información suficiente sobre la herramienta para que un cliente pueda armar una solicitud de ejecución; en el caso de la acción *index* se devuelve una estructura JSON que contiene un arreglo con la información esencial de cada herramienta, suficiente para que la aplicación que lo solicita le muestre al usuario un listado de las herramientas disponibles en la plataforma y una descripción breve de cada una.

Podemos considerar al servicio *Tools* como un traductor entre los archivos de configuración de cada programa en la plataforma y las estructuras de datos en JSON con que trabaja la plataforma, como lo podemos ver en la figura 4.3.

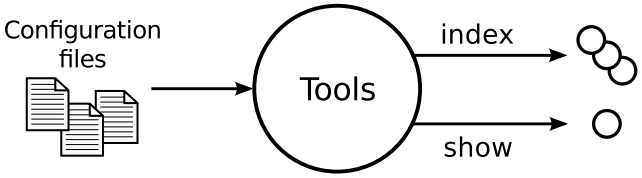

Figura 4.3: Generación del servicio *Tools* basado en los archivos de configuración

Una vez que un usuario elige los parámetros de ejecución para una herramienta, el programa que use —por ejemplo el frontend de la plataforma— deberá completar el objeto JSON proporcionado por el servicio *Tools* con esta información y enviarlo de vuelta a la plataforma para su ejecución. El encargado de procesar esta solicitud es el servicio *Jobs*.

Jobs

El servicio *Jobs* se encarga de recibir un objeto JSON que representa una de las herramientas incluidas en la plataforma, con suficiente información para su ejecución, y generar el script de ejecución que será enviado a PBS.

Jobs ofrece cuatro métodos: *create* para generar un nuevo job en el cluster; *show* para consultar un job generado anteriormente, y *destroy* para borrar los archivos del cluster relacionados a un job.

Este servicio realizará las siguientes funciones:

- Compara el objeto JSON recibido contra los esquemas de validación(sección **??**).
- Genera un identificador numérico único para el job y crea un directorio en el cluster con ese identificador como nombre.
- Genera un script con la línea de comando que ejecutará el programa y las opciones especificadas en el objeto JSON, tales como adiciones a la variable de entorno *PATH*, así como parámetros para la ejecución en el cluster como número de nodos, núcleos y cola de ejecución a utilizar, ya sea que esta información venga del objeto JSON proporcionado por el usuario o de la configuración de la herramienta.
- Llama al comando `qsub` de PBS para enviar el script generado a una cola de ejecución.
- Devuelve un objeto JSON con el número de identificador generado para este job.

Workflows

Un *workflow* consistirá de una lista de jobs y sus relaciones entre ellos. Los métodos que este servicio ofrece son los mismos que el servicio *Jobs*. Su método de ejecución se describe en la sección 4.2.

4.1.2. Frontend

El frontend cumple tres funciones principales en la plataforma:

- Ser un intermediario entre el usuario y el backend; y a través de él ejecutar experimentos in silico en el cluster.
- Proveer a los usuarios un fácil acceso a los archivos de datos generados por sus experimentos.
- Ser la interfaz mediante la cual se manipulan opciones de la plataforma, como la administración de los usuarios, definición de formatos de archivo para los experimentos, etcétera.

Como intermediario entre el usuario y el backend, el frontend permite al usuario tanto ejecutar un programa en el cluster de manera individual como definir un workflow que se forma de la ejecución de varios programas. En la figura 4.4 podemos ver como el frontend consume el servicio Tools, recibiendo un objeto JSON que representa alguna de las herramientas en la plataforma, y a partir de este objeto construye y despliega un formulario HTML que al enviar al servicio Jobs, este se encargará de preparar su ejecución en Mazorka. En la figura 4.5 podemos ver el escenario en el que un usuario crea un workflow. Para esto utiliza el editor visual, que igualmente consume el servicio Tools para obtener información sobre los programas que compondrán el workflow. Una vez diseñado el workflow, el editor envía la solicitud de creación al servicio workflows, que a su vez se encargará de prepararlo para su ejecución en Mazorka.

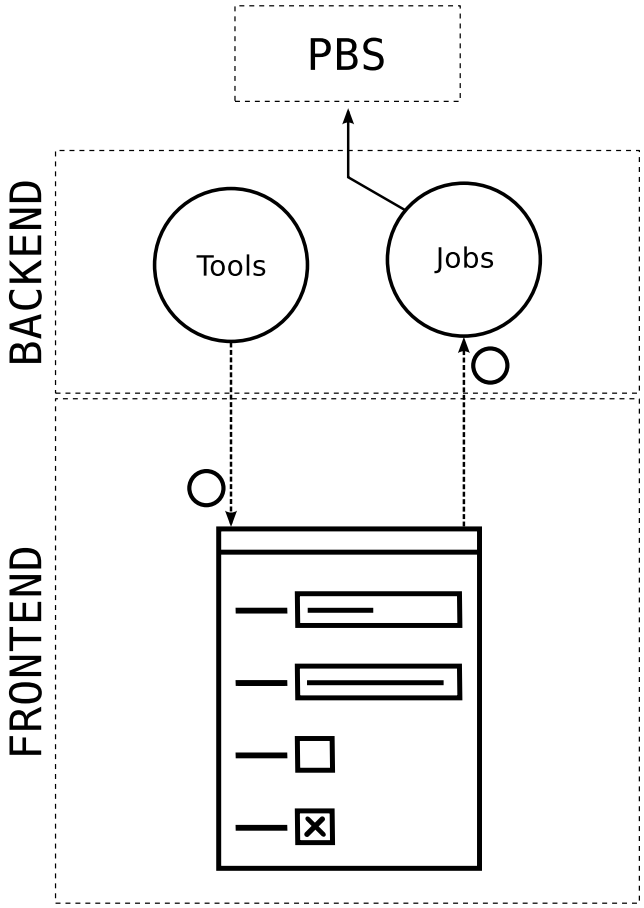

Figura 4.4: Proceso para la ejecución de un job individual

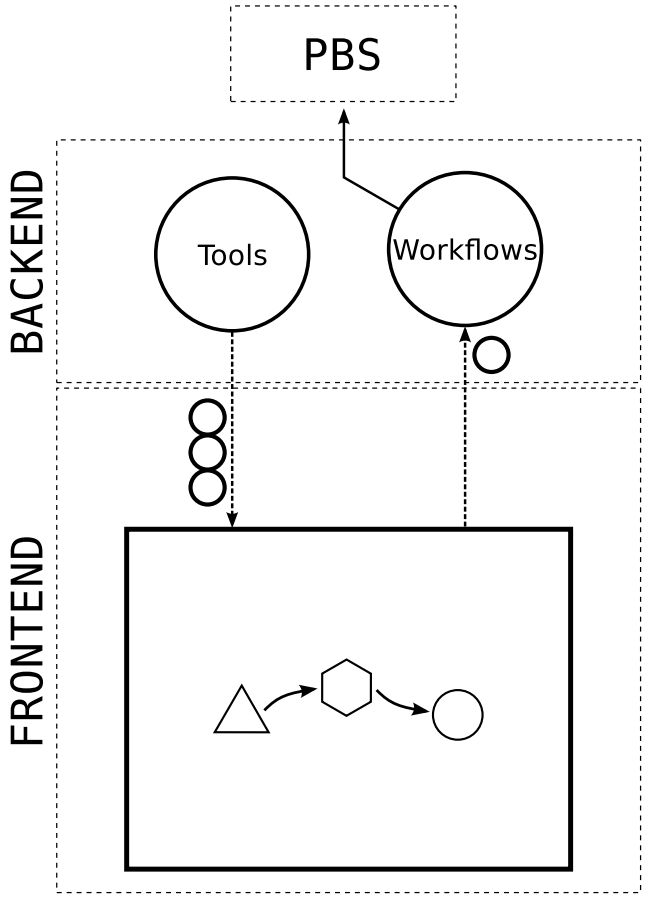

Figura 4.5: Proceso para la ejecución de un workflow

4.2. Demonio

Existen varias tareas que la plataforma debe realizar adicionales a las acciones realizadas inmediatamente después de que el usuario envíe un job o workflow a ejecución. Entre estas está consultar al administrador de colas de Mazorka para actualizar el estado los jobs enviados a través de la plataforma que se encuentren en ejecución. Para esto agregamos a la plataforma un servicio o *demonio* mediante la extensión de Rails *daemons-rails*.

Este demonio es simplemente un proceso que se ejecuta de manera no-interactiva y con determinada frecuencia —por defecto un minuto— y las funciones que cumple son:

- Buscar en la base de datos los jobs que fueron lanzados desde la plataforma y cuyo estatus sea *en cola* o *ejecutándose*.
- Consultar al administrador de colas del cluster para conocer el estatus actual de estos jobs.
- Si el estatus es distinto al registrado en la base de datos, actualizarlo.

- Buscar en la base de datos los workflows cuyo estatus sea *ejecutándose* y si los jobs que estaba esperando han terminado satisfactoriamente, entonces ejecutar los jobs que dependían de estos o en su defecto actualizar el estatus del job de acuerdo al estatus de los jobs que lo componen.

4.3. Acceso directo a archivos

Para hacer posible la construcción de workflows en la plataforma es necesario que podamos describir a detalle las entradas y salidas de un programa. Para esto creamos dos secciones en el archivo de configuración, llamadas *input* y *output*, donde podemos definir qué tipo de entrada recibe (flujo estándar de entrada, ruta a un archivo, etcétera) y en qué formato; así como qué salidas genera (flujo estándar de salida, archivo de salida, etcétera) y en qué formato. En el código incluido en 5.1 podemos observar un ejemplo de estas dos secciones.

Los archivos los identificaremos de dos maneras. La manera principal de identificarlos será mediante un UUID, que se podrán utilizar para descargar un archivo o especificarlo como recurso dentro de un job como *http://olote/api/files/UUID*. Los UUID serán generados automáticamente por la plataforma en los siguientes casos:

- Todos los archivos subidos a la plataforma por el usuario.
- Todos los archivos generados directamente por la plataforma, como los scripts que son enviados a ejecución.
- Todos los archivos generados por un job en el cluster cuando en la configuración de la herramienta exista una definición para ese archivo en la sección *outputs*.

Cuando un archivo de salida no esta definido en la configuración de una herramienta, aún es posible accederlo especificando el recurso del job que lo generó y la ruta del archivo partiendo del directorio donde se ejecutó el programa. Por ejemplo, si el job con id 719 genera un archivo de salida *output/1.pdb*, entonces podremos descargarlo y especificarlo como recurso como *http://olote/jobs/719/output/1.pdb*. Esto es importante cuando deseamos usar una archivo como entrada en un workflow pero no se generó un UUID para él.

4.4. Retroalimentación del consumo de recursos

Cuando un programa es ejecutado, este consumirá dos recursos de cómputo principales: ciclos del procesador y memoria. Cuando el programa toma un tiempo de ejecución considerable, como es el caso de los programas que se ejecutarán a través de la plataforma, es importante poder estimar la cantidad y proporción de recursos que consumen. La razón principal de esto es que Mazorka es un cluster heterogéneo, es decir, no todos sus nodos tienen las mismas características. En particular, existen nodos con mucha más memoria y grupos de nodos con una mayor cantidad de

núcleos. Estos nodos están agrupados de manera lógica mediante colas en el cluster. Esto significa que un job puede ser enviado a una cola en la que el recurso principal es la disponibilidad de memoria, una cola cuyos nodos tienen un número alto de núcleos o, una cola con un balance de ambos.

Si se sabe que un programa utiliza mucha memoria o es altamente paralelizable, entonces su tiempo de ejecución puede verse reducido considerablemente si es enviado a la cola adecuada. Es posible predecir en cierto grado el uso de recursos de un programa, ya sea analizando el algoritmo que utiliza y los datos de entrada, o simplemente de acuerdo a la clase de programa que es (e.g. los programas que realizan ensambles de secuencias biológicas requieren de una enorme cantidad de memoria). Sin embargo, el uso final de recursos puede variar mucho de acuerdo a ciertos parámetros de ejecución del programa o distintos datos de entrada. Además, los usuarios del cluster no siempre tienen el entrenamiento necesario para hacer este tipo de estimación. Por estas razones es importante que los usuarios tengan manera de monitorear el consumo de recursos del cluster por sus programas.

Aunque es posible simplemente proporcionarles el identificador del proceso y que ellos utilicen programas del sistema operativo como *top* para realizar este análisis, se buscó explotar para este fin la información proporcionada por la plataforma Ganglia. Ganglia es un sistema de monitoreo para equipos de alto rendimiento como clusters y ya se encuentra instalado en Mazorka.

4.4.1. Asignación automática de colas

A cada programa que se agregue a la herramienta se le podrán definir opciones de ejecución en el cluster como cola, número de nodos y número de núcleos por nodo a usar por defecto. Cuando estas opciones no se especifiquen, la plataforma le presentará estas opciones al usuario para que él las defina.

Capítulo 5

Modelos de datos

La base de la plataforma son los programas que se ejecutarán a través de ella, por lo que la manera en que se definen y configuran es parte esencial del proyecto.

El tipo de programa con el que trabajará la plataforma es un programa que ejecuta sobre un sistema operativo de tipo Unix desde mediante una línea de comando y ejecuta de manera no-interactiva. Sus datos de entrada provienen del flujo estándar de entrada o desde archivos que se pueden especificar como parámetros en la línea de comando que los invoca. Los datos que genera son enviados al flujo estándar de salida o a archivos, ya sea que se especifiquen en la línea de comando o que se guarden en el directorio donde el programa fue ejecutado. Se considera solo este tipo de programa porque son estos los que se ejecutan actualmente en el cluster.

Las consideraciones principales que se tomaron en cuenta para elegir un método de representación fueron:

- La cantidad de programas a ejecutarse a través de la plataforma puede cambiar conforme cambien las necesidades de los usuarios y las tendencias en el área de la bioinformática.
- La configuración de un programa puede evolucionar debido a un cambio de versión o a que se necesita presentar a los usuarios un conjunto mayor de opciones.
- Cada programa tiene un número y tipo de parámetros distinto, por lo que se requiere flexibilidad en su representación.
- Es deseable que la configuración de cada programa se guarde en un archivo de texto, tanto para tener acceso a las ventajas de un software manejador de versiones como para facilitar su copia y transferencia a diferentes instalaciones de la plataforma.

En base a esto se eligió representar cada programa mediante un archivo en formato YAML que podrá ser importado directamente a la plataforma. Esto presenta las siguientes ventajas:

- Modificar la configuración de la herramienta es tan fácil como editar el archivo de texto.

- Los archivos tienen un formato estructurado y claro que es posible leer y modificar con un editor de texto.
- Este formato es suficientemente flexible para tener configuraciones muy distintas entre diferentes programas.

5.1. Estructura de los archivos de configuración

Los archivos de configuración de cada herramienta están divididos en seis secciones. Estas secciones están delimitadas por el título de la sección seguido de dos puntos y cada opción dentro de esta sección tiene un margen a la izquierda, como lo dicta el estándar YAML. En el código ejemplo 5.1 mostramos un ejemplo básico de la definición del programa BLAST.

5.1.1. Información

Esta sección contiene información general sobre la herramienta. Esta formada por los campos *short_name*, *long_name*, *description* y *help*.

5.1.2. Icono

Los datos de esta sección se usarán únicamente en el editor visual para dibujar un icono que representará dentro de un workflow la ejecución de este programa. Está formada por los campos *path* y *fill*.

5.1.3. Ambiente de ejecución

En esta sección se definen opciones que afectan el ambiente de ejecución del programa. Esta formado por el campo *executes_in* que define si el programa será enviado al administrador de ejecución PBS, que será el caso en todos los programas a ejecutarse en Mazorka; el campo *binary* que indica la ruta del archivo ejecutable, ya que los programas utilizados en Mazorka no siempre están incluidos en el path del sistema; el campo *path_requires* que utilizaremos para agregar directorios al path en caso de ser necesario; y el campo *queue* que definirá, si el campo esta lleno, en qué cola de PBS deberá ejecutarse el programa y, en caso de estar vacío, dejarle al usuario la elección de la cola de ejecución.

5.1.4. Entradas

Esta sección esta dividida en dos subsecciones: *stdin* y *files*. *stdin* define las opciones para el flujo estándar de entrada del programa mientras que *files* es un arreglo que contiene opciones para cada archivo de entrada que use el programa. En el caso de *stdin* las únicas opciones que requieren son el formato de los datos que recibe, en caso

de utilizar la entrada estándar, y especificar si esta se requiere. Las opciones para la lista de archivos de entrada son el formato del archivo, el parámetro de la línea de comando que se usará para especificar el archivo y si este archivo es requerido.

5.1.5. Opciones

Esta será normalmente la sección de mayor tamaño en la configuración de una herramienta. Aquí se definen cada uno de los parámetros de la línea de comando que se quieran dar como opción para la ejecución del programa. Los diferentes tipos de opciones disponibles se explican más adelante en su propia sección.

5.1.6. Salidas

Cualquier archivo de salida que se genere en el directorio donde se ejecutó el programa podrá ser descargado por el usuario y usado como entrada dentro de un workflow a través del frontend de la plataforma, sin embargo los archivos de salida que aquí se especifiquen serán los directamente accesibles desde la página de resultados.

Código 5.1: Configuración de un programa en la plataforma

```
# blast
---
info:
  short_name: Blast
  long_name: Blast
  description: "Algorithm for comparing primary ..."
  help: "Use this program to ..."

icon:
  path: "M 0,0 36,0 55,28 37,55 l -37,0 z"
  fill: Aquamarine

environment:
  executes_in: pbs
  binary: /data/storage/software/blast-2.2.26/bin/blastall
  path_requires: nil
  queue: biofis

inputs:
  stdin:
  files:
    - label: Sequences to align
      format: fasta
      modifier: "-i "
      required: true

options:
  - label: Database
    description: Select a database
    type: select
```

```
    select_function: blastDB
    modifier: "-d "
    required: true

  - label: Alignment type
    description: Here you can define the type of alignment
    modifier: "-p "
    type: select
    select_list:
      - value: blastn
        label: Blastn
      - value: blastp
        label: Blastp
      - value: blastx
        label: Blastx
```

5.2. Opciones de un programa

En la Tabla 5.1 se listan las opciones que se pueden especificar en la sección *options* del archivo de configuración, junto con los valores aceptados para cada una de ellas. Las expresiones regulares están expresadas con la notación de Ruby.

5.3. Representaciones orientadas al contexto

La abstracción descrita arriba contiene, en un solo lugar, toda la información que deseamos almacenar sobre un programa, sin embargo en ningún caso requerimos manejar o proporcionar esta información en su totalidad al mismo tiempo. Internamente, la plataforma proporciona información sobre los programas a ejecutar tanto a otras partes del programa como al mundo exterior mediante servicios. A partir de la descripción completa del programa se generan diferentes representaciones, dependiendo del uso que se le dará a la información.

Un ejemplo claro de esto lo podemos observar en las respuestas a dos de las funciones de la API. La primera, cuya respuesta se incluye en el listado 5.2, es una compilación de los programas disponibles en la plataforma pero la información proveída sobre cada uno se limita a la necesaria para generar un menú, localizar el recurso completo (mediante el valor de *location*) y, en el caso del editor visual del frontend Web, poder dibujar el icono correspondiente al programa. La segunda respuesta, incluida en el listado 5.3, contiene información sobre los parámetros de ejecución de un programa suficiente para que un cliente pueda preparar una solicitud de ejecución del programa a la plataforma. En este caso se incluye más información que en la respuesta al índice de aplicaciones, pero se elimina el campo *modifier*, al cual solo debe tener acceso la plataforma y no el cliente.

Campo	Descripción	Valores permitidos
label	Nombre de la opción	`/^[a-zA-Z0-9]+[a-zA-Z0-9 ]*$/`
description	Descripción de la opción: como modifica la ejecución, su impacto en el sentido biológico, etcétera	`/^[a-zA-Z0-9 ]*$/`
help	Mensaje que indica el tipo de datos que la opción recibe	`/^[a-zA-Z0-9 ]*$/`
modifier	El modificador es la cadena que será insertada antes del valor que se le haya dado a la opción	`/^[a-zA-Z0-9 ]*$/`
required	Define si esta es una opción obligatoria	`/^(true|false|)$/`
type	Define el tipo de opción. El valor de este campo le indicará a la plataforma si debe buscar alguno de los campos siguientes	`/^(text|number|select|flag)$/`
pattern	Cuando *type* es igual a *text* el valor de este campo es una expresión regular que indica los valores aceptados	Expresión regular que funcione en Ruby y Javascript
max	Cuando *type* es igual a *number* el valor de este campo indica el valor máximo que la opción puede tomar	`/^[0-9]$/`
min	Cuando *type* es igual a *number* el valor de este campo indica el valor mínimo que la opción puede tomar	`/^[0-9]$/`
step	Cuando *type* es igual a *number* el valor de este campo indica los incrementos posibles que la opción puede tomar	`/^[0-9]$/`
select_list	Cuando *type* es igual a *select* este campo debe contener un arreglo de diccionarios, donde cada diccionario contiene un elemento *value* y un elemento *label*	Arreglo de diccionarios con los campos *value* y *label*
select_function	Cuando *type* es igual a *select* este campo contiene el nombre de una función que proveerá un arreglo como el mencionado en *select_list*	Nombre de una función

Tabla 5.1: Campos válidos en las opciones de un programa

Código 5.2: Representación generada por la acción *index*

```
blastall:
  name_menu: Blast
  name_title: Blast
  description: Algorithm for comparing primary biological ...
  icon:
    path: M 0,0 36,0 55,28 37,55 l -37,0 z
    fill: Aquamarine
  location: http://pasteur:3000/api/v1/tools/blastall
phyml:
  ...
trinity:
  ...
```

5.4. Uso de esquemas para validación

Para varias operaciones de la plataforma es importante garantizar que la estructura de datos que se procesa o se recibe cumple con ciertas condiciones. Aunque es posible escribir rutinas que busquen la presencia y validez de ciertos campos en cada caso particular, es más conveniente utilizar un método estándar. En el caso de información en JSON existen los llamados *esquemas de JSON*, que aunque de momento son un borrador de la IEEE, cuentan con la suficiente popularidad y disponibilidad de herramientas para que su uso sea viable.

En el listado 5.4 se muestra una parte del esquema que valida la estructura de 5.1.

Código 5.3: Representación generada por la acción *show*

```
---
name_short: Blast
name_long: Blast
description: Algorithm for comparing primary biological...
icon:
  path: M 0,0 36,0 55,28 37,55 l -37,0 z
  fill: Aquamarine
input:
  - stdin: null
  - input_sequence:
    id: input_sequence
    format: fasta
    required: true
options:
- Database:
  - id: database
    label: Database
    type: select
    select_list: [...]
    required: true
- Alignment options:
  - id: alignment_type
    label: Alignment type
    type: select
    select_list:
    - value: blastn
      label: Blastn
    - value: blastp
      label: Blastp
    - value: blastx
      label: Blastx
```

Código 5.4: Esquema JSON para la validación de un programa

```
title: Schema for program description
type: object
$schema: http://json-schema.org/draft-03/schema
properties:
  name_long:
    type: string
    required: true
  name_short:
    type: string
    required: true
  binary:
    type: string
    required: true
  description:
    type: string
    required: true
  icon:
    type: object
    required: true
    properties:
      fill:
        type: string
        required: true
      path:
        type: string
        required: false
  input:
    type: array
    required: true
    items:
    - type: object
      required: false
      properties:
        stdin:
          type: string
          required: true
    - type: object
      required: false
      properties:
        format:
          type: string
          required: false
        input_sequence:
          type: 'null'
          required: false
        modifier:
          type: string
          required: false
        required:
          type: boolean
          required: true
```

Capítulo 6

Implementación

A partir del análisis, diseño y definición del modelo de datos usado en la plataforma podemos ver algunos de los detalles de su implementación. Comenzaremos por describir los lenguajes de programación y plataformas de desarrollo elegidas para después comentar la manera en que se construyeron los elementos de la plataforma a partir de los conceptos definidos en el diseño.

6.1. Plataformas de desarrollo

Para el desarrollo de la plataforma se utilizaron los lenguajes de programación Ruby y Javascript, junto con la plataforma de desarrollo Web Ruby on Rails. El backend de la plataforma esta implementado totalmente en Rails y el frontend es una mezcla de Rails y Javascript, donde este último fue utilizado para la generación de los formularios Web de las herramientas, la generación del editor visual y en general la interactividad del lado del cliente. En la figura 6.1 podemos observar la relación entre las tecnologías de desarrollo y las capas de la plataforma. Podemos observar también que cuando se requiere obtener información en el cliente sin solicitar una página completa al servidor, utilizamos llamadas AJAX.

6.2. Espacios de nombres y variables

A la agrupación abstracta de la funcionalidad en el diseño de la plataforma le corresponde una organización análoga en el agrupamiento de variables, clases y otros elementos en la implementación. Debido a que la implementación de la plataforma esta dividida en código del lado del servidor en Ruby on Rails y código del lado del cliente en Javascript, es importante mencionar este agrupamiento en ambos.

Ruby cuenta con la funcionalidad de especificar espacios de nombres y las variables que se creen dentro de estos son accesibles directamente solo para el código que se encuentre en el mismo espacio de nombres. Si se desean acceder desde código fuera de ese espacio de nombres, deben especificar la ubicación completa de la variable.

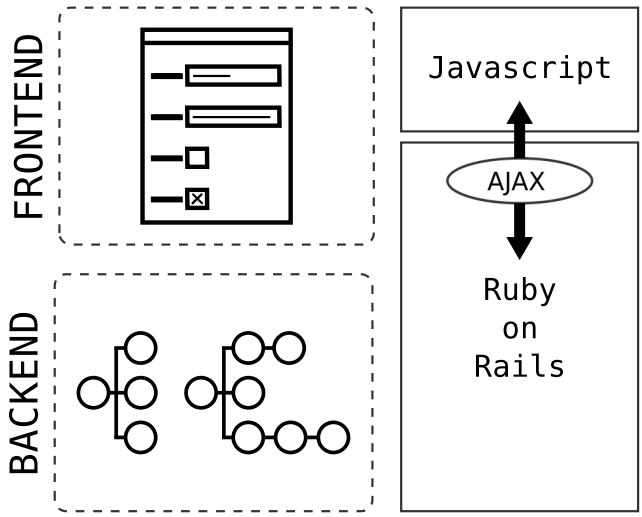

Figura 6.1: Lenguajes y plataformas de desarrollo utilizadas por capa

Por ejemplo, si se define el espacio de nombres *Foo* y dentro de este la variable *bar*, entonces debemos utilizar el nombre *Foo::bar* desde cualquier espacio de nombres distinto a *Foo*.

Utilizamos primeramente el espacio de nombres *Olote* para todas las variables relacionadas con la plataforma. Dentro de este espacio de nombres anidamos dos espacios de nombres llamados *Api* y *Frontend* para todas las variables relacionadas con el backend y el frontend de la aplicación, respectivamente. Esto es importante porque ambas capas de la plataforma trabajan con los mismos conceptos, aunque a un nivel distinto de funcionalidad, lo que lleva a que en ambas capas existan clases que representen conceptos como *Tools*, *Jobs* y *Workflows*. Esto nos lleva a que existan las mismas clases dentro de ambos espacios de nombres, por ejemplo: *Olote::Api::Job* y *Olote::Frontend::Job*.

Javascript no cuenta con espacios de nombres, por lo que utilizamos el tipo de datos que en este lenguaje se conoce como un objeto y dentro de este objeto anidamos otros objetos. Para crear algo equivalente a un espacio de nombres al del ejemplo en Ruby mostrado en el párrafo anterior, creamos el objeto *Olote* y dentro de este los objetos *Api* y *Frontend*, de manera que podemos tener variables llamadas *Job* en cada uno: *Olote.Api.Job* y *Olote.Frontend.Job*.

6.3. Recursos

Siguiendo las convenciones del estilo REST, definimos la plataforma en base a los siguientes recursos, algunos de los cuales son la base directa para la definición de los servicios ofrecidos por el backend.

- *Tools*: representa alguno de los programas que pueden ser ejecutados desde la plataforma.

- *Jobs*: representa las opciones y archivos generados por la ejecución de uno de estos programas.
- *Workflows*: representa un conjunto de jobs.
- *Files*: representa los archivos de datos utilizados como entrada para un job o generados como salida por este.
- *Formats*: representa los formatos de archivo que conoce la plataforma.
- *Users*: representa a los usuarios de la plataforma y serán los dueños de los jobs, workflows y archivos.

La interacción con cada uno de estos recursos se realiza mediante un conjunto de acciones sobre ellos de entre siete acciones posibles: *index*, *new*, *create*, *show*, *edit*, *update* y *destroy*. Las acciones disponibles para un recurso determinan como es que se puede interactuar con ellos. Algunos de estos recursos se encuentran tanto en el backend como en el frontend. La tabla 6.1 contiene una lista con las acciones disponibles para cada uno de estos recursos.

Recurso	**URI**	**Acciones**
Api::Tools	/api/tools	Index, Show
Api::Formats	/api/formats	Index, Show
Api::Files	/api/files	Index, Show
Api::Jobs	/api/jobs	Create, Show, Destroy
Api::Workflows	/api/workflows	Create, Show, Destroy
Frontend::Tools	/tools	Index, Show
Frontend::Jobs	/jobs	Index, New, Create, Show, Edit, Destroy
Frontend::Workflows	/workflows	Index, New, Create, Show, Edit, Destroy
Users	/users	Index, New, Create, Show, Edit, Update, Destroy

Tabla 6.1: Recursos de la plataforma

6.3.1. Hipermedia

En REST, las referencias entre recursos deben seguir la práctica de la *hipermedia como el motor del estado de la aplicación*. Esto significa que si dentro de un recurso existe una referencia a otro recurso, esta referencia debe ser *hipermedia*, es decir, que la referencia debe indicar la ubicación completa del recurso mediante un URI. Un ejemplo de esto en la plataforma lo podemos ver en el caso de la acción *index* del recurso *Tools*, que devolverá un objeto JSON que representa un arreglo con información sobre los programas disponibles en la plataforma. Uno de estos campos de información indica la ubicación de ese recurso y para que esta se considere hipermedia debemos incluir su URI completo. Si en la plataforma han sido agregados los programas *Blast* y *PhyML*, la acción *index* devolverá como parte de su respuesta las URIs *http://olote/api/tools/blast* y *http://olote/api/tools/phyml*.

En el caso de un objeto que representa un job, este hará referencia a distintos archivos de entrada. Esta referencia deberá ser un URI que indique la ubicación de tal recurso. Así mismo, esto se aplica en el caso de los objetos que representan un workflow; las referencias que haga a los jobs que los componen deberán incluir el URI completo de tal recurso.

6.3.2. Transacciones sin estado

Las transacciones sobre los recursos de la plataforma se realizan sin almacenar un estado. En este párrafo describiremos el recurso *Jobs*, pero el mismo principio aplica para los demás recursos. Cuando un cliente ejecuta la acción *create* del recurso *Jobs* — mediante la URI *http://olote/api/jobs* y el método POST de HTTP— con un objeto JSON que representa un job válido, la plataforma realizará la acción correspondiente, que es generar los archivos del Job en el cluster y enviarlo a la cola de ejecución. La plataforma no necesita de ninguna transacción previa ni una transacción posterior para finalizar el envío del Job. La acción es independiente y atómica.

6.3.3. Controladores

Se crea un controlador en Rails para cada uno de los recursos y dos controladores adicionales: *StaticPages* para la generación de páginas estáticas como la página principal de la plataforma y *Sessions* para la administración de sesiones de usuario. En la figura 6.2 podemos ver la organización de los controladores en la plataforma.

6.4. Rutas

A partir de la definición anterior de los recursos de la plataforma y sus acciones podemos construir las rutas de la aplicación. En Rails, las rutas son definidas en el archivo *config/routes.rb*, donde cada ruta tiene el formato:

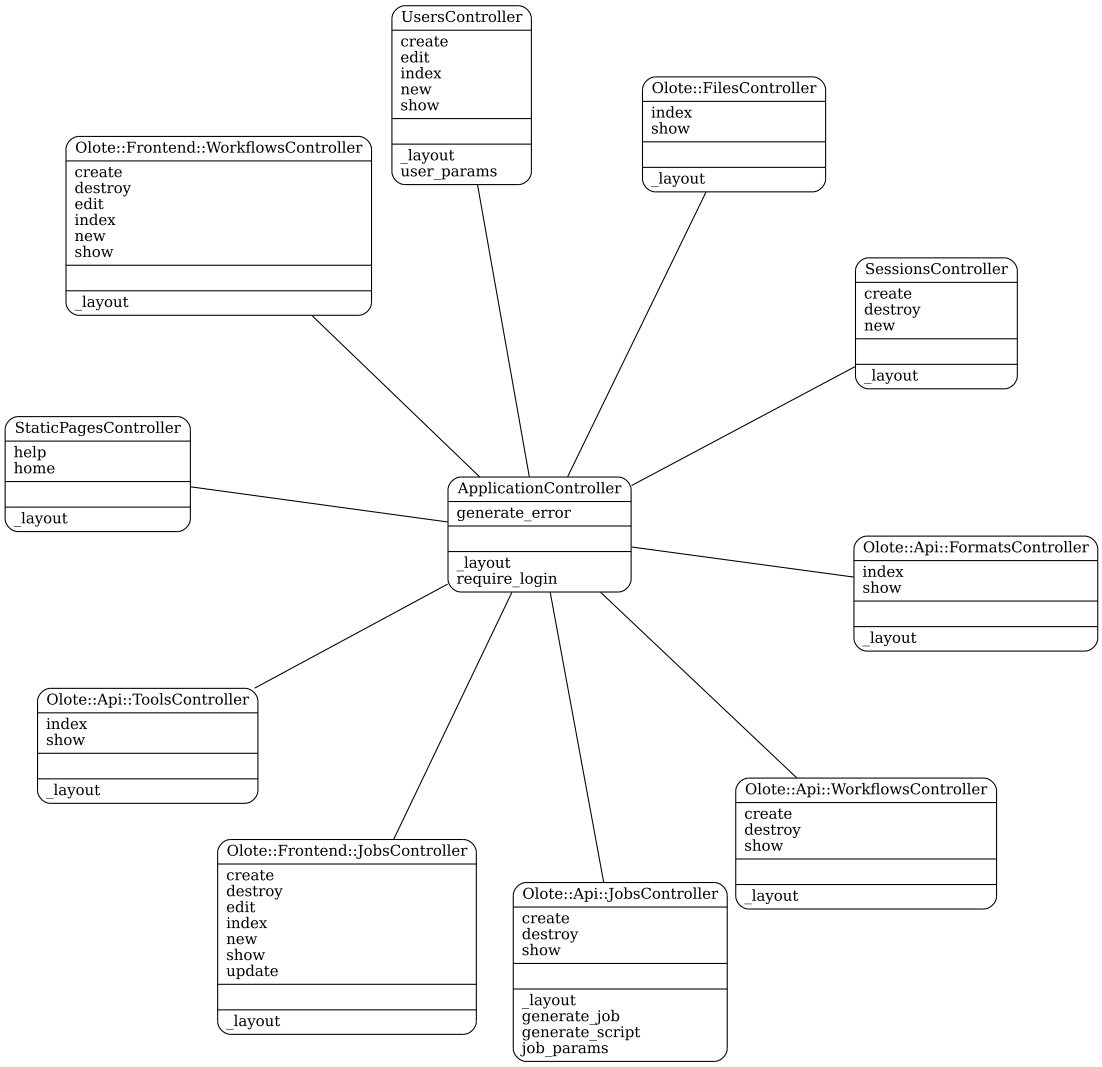

Figura 6.2: Controladores de la plataforma

`match '/`*ruta*`', to: '`*controlador*`#`*accion*`', via: '`*metodo*`',` *parametros*

Rails ofrece una manera compacta de definir rutas para las acciones de recursos REST mediante el comando *resources*. Este comando crea automáticamente rutas para las siete acciones de REST, pero se pueden especificar menos mediante los parámetros *only* y *except*. De esta manera generamos el archivo de rutas que se incluye en el código 6.1.

Código 6.1: Rutas del sistema

```
scope module: 'olote' do
  namespace :api do
    match "/tools/:id/:version", to: "tools#show", via: 'get'

    resources :tools,     only: [:index, :show]
```

```
    resources :files,     only: [:show]
    resources :formats,   only: [:index, :show]
    resources :jobs,      only: [:create, :show, :destroy]
    resources :workflows, only: [:create, :show, :destroy]
  end

  scope module: 'frontend' do
    match "/jobs/new/:tool/:version", to: "jobs#new", via: 'get'
    match "/jobs/new/:tool",          to: "jobs#new", via: 'get'

    resources :jobs,       except: [:new, :update]
    resources :workflows,  except: [:update]
  end
end

root 'static_pages#home'

resources :users
resources :sessions, only: [:new, :create, :destroy]
match '/signin',     to: "sessions#new",      via: 'get'
match '/signout',    to: "sessions#destroy",  via: 'delete'
match '/users/new',  to: "users#new",         via: 'get'
```

6.5. Editor visual de workflows

Para la construcción del editor visual de workflows utilizamos la extensión de jQuery llamada Raphaël y generamos dos contenedores; uno para contener el menú y otro para el workflow. Primero se llena el contenedor del menú mediante la acción *index* del servicio *Tools* en el backend. Conforme el usuario agregue programas al workflows, generaremos una copia del icono del programa en el contenedor del workflow. Cuando el usuario haga un clic en un icono del workflow, utilizaremos la acción *show* del mismo servicio para obtener todos los detalles del programa y generar un formulario Web con los parámetros de ejecución. En la figura 6.3 se muestra un diagrama general del funcionamiento del editor.

6.6. Demostración

En la Figura 6.4 mostramos el formulario que resulta a partir del archivo de configuración completo similar al del Código 5.1. Ahí podemos observar como las opciones del archivo de configuración serán traducidas a elementos de entrada en un formulario, donde el tipo de la entrada corresponde al valor que se haya elegido en el campo *type*. Si se guardó información en los campos *description* o *help* esta se mostrará como un *tooltip* al colocar el puntero encima del signo de interrogación junto a los campos de entrada.

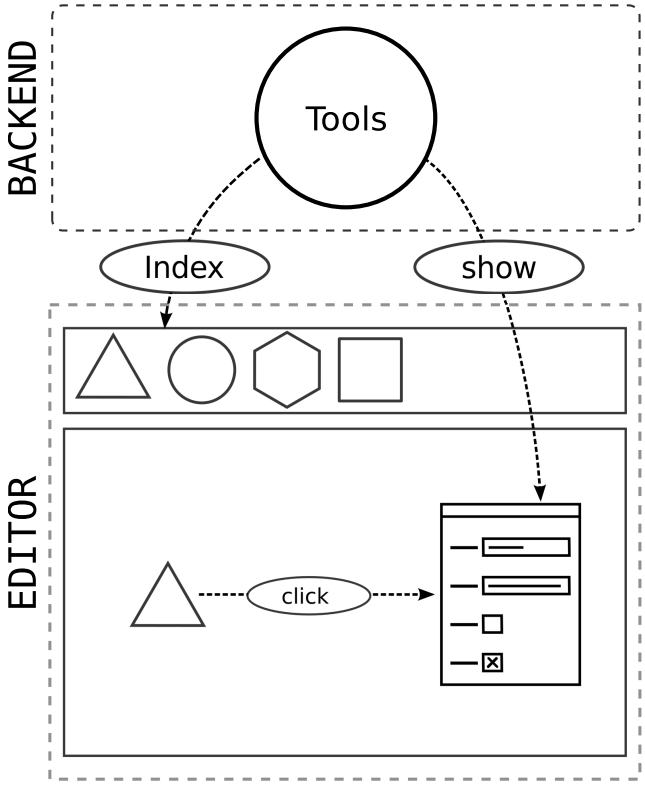

Figura 6.3: Relación entre el editor visual y la capa de servicios

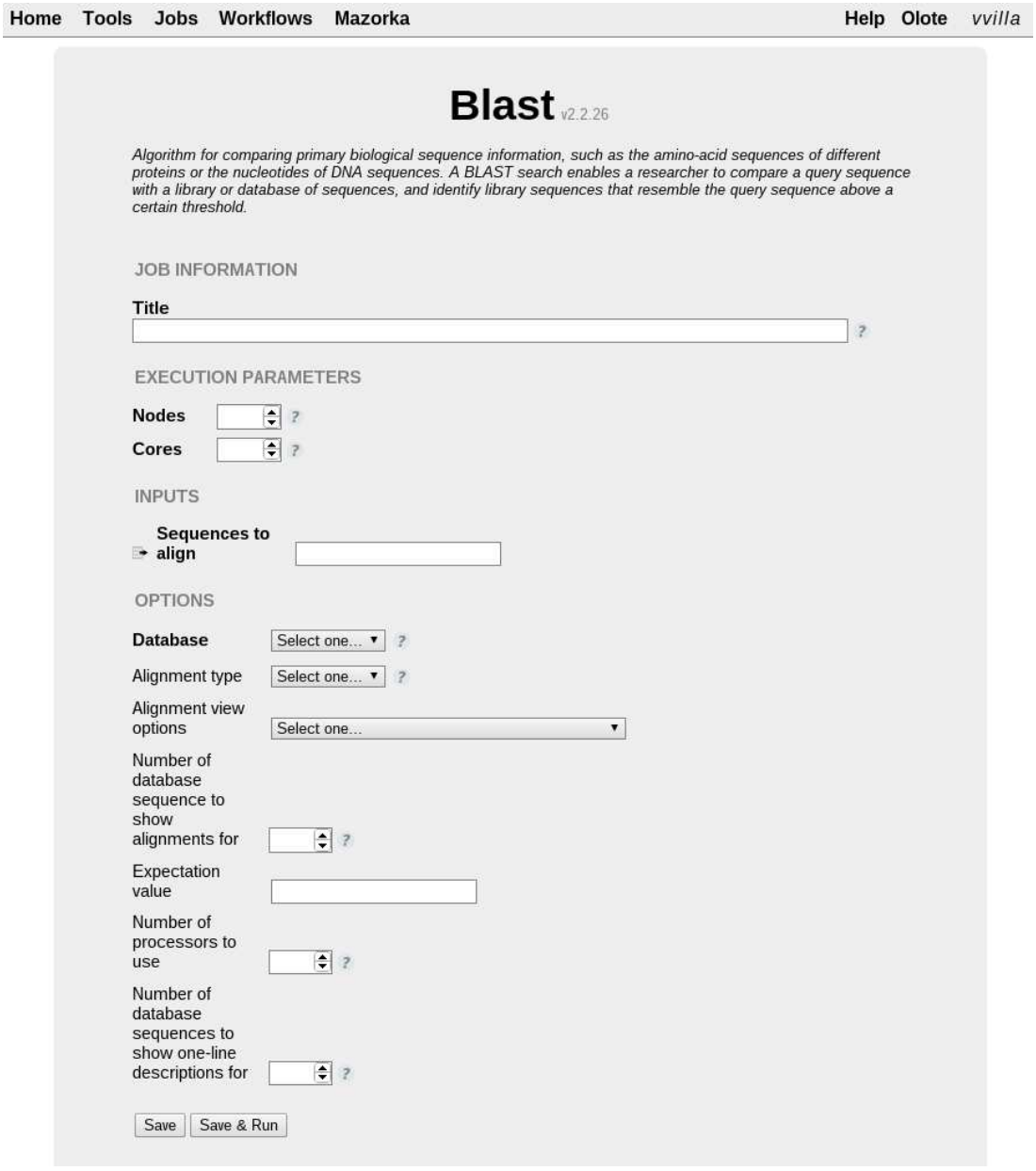

Figura 6.4: Configuración de parámetros para la ejecución de un programa

Capítulo 7

Resultados y conclusiones

Aunque el desarrollo de la plataforma aún no está terminado, se tiene un avance suficiente para que sea de utilidad en el Langebio y los suficientes resultados para extraer conclusiones.

7.1. Resultados

Partiendo de los objetivos particulares mencionados en la sección 1.2.3, este proyecto ha logrado los siguientes resultados:

- La definición de los modelos de datos que representan programas, jobs y workflows a ejecutar en el cluster.
- La implementación de una plataforma Web en Ruby on Rails que implemente servicios Web basados en esos modelos de datos.
- El mecanismo de ejecución de jobs en Mazorka a partir de estos servicios Web.
- El mecanismo para la identificación y descarga de archivos a través de UUID, tanto subidos a la plataforma como generados por los jobs.
- La implementación de un frontend Web que construya formularios Web a partir de la representación de un programa.
- El mecanismo para que el frontend genere la representación de un job de acuerdo a los parámetros definidos por el usuario en el formulario.
- Un editor de workflows en el frontend que permite definir workflows para ejecutarse en el cluster de manera visual.

7.2. Discusión

El modelo de datos que se definió permite configurar de manera sencilla los programas a ejecutar en Mazorka y a partir de este construir modelos de datos que describan jobs y workflows. Los servicios Web que se construyeron a partir de estos permiten tener una API clara que puede ser utilizada por programas externos.

El uso de una interfaz visual para la definición de experimentos in silico utilizando funcionalidad HTML existente y estándar permite que los usuarios de la plataforma interactúen de una manera familiar con los programas que antes tenían problemas utilizando.

7.3. Conclusiones

Es posible construir servicios claros y reutilizables a partir de un modelo de datos sencillo. Es imperativo mantener modelos de datos sencillos cuando estos serán construidos y modificados por personas no dedicadas al desarrollo de aplicaciones.

Si la interfaz gráfica de una aplicación se apega a estándares y herramientas bien establecidas, esta será más fácil de usar para un usuario y más estable.

El diseño de la plataforma como una capa intermedia delgada entre el usuario y el cluster contribuye a que sea conveniente, porque aumenta el control sobre los recursos computacionales sin romper completamente el esquema de uso. Este tipo de diseño también permite que sea reutilizable.

En nuestra opinión, el desarrollo de esta plataforma demuestra que el uso de una herramienta visual para la ejecución de experimentos en un cluster de cómputo contribuye al mejor uso de los recursos de cómputo por parte de los usuarios.

7.4. Trabajo futuro

Las siguientes son las acciones que creemos pueden mejorar la utilidad de la plataforma.

- Permitir la creación de protocolos a partir de workflows, los cuales puedan compartirse entre los usuarios para ejecutar experimentos in silico estandarizados.
- Modificar la plataforma para que pueda ejecutar los elementos de un workflow en clusters distintos.
- Habilitar la generación de servicios Web XML en el backend a partir de los servicios ya existentes en JSON. Generar descripciones WSDL a partir de los esquemas JSON.
- Habilitar el frontend para el consumo y ejecución de servicios web en XML.

Bibliografía

[1] Max-Planck Institute for Developmental Biology. Bioinformatics toolkit. `http://toolkit.tuebingen.mpg.de/`. Consultada: 2013-02-21.

[2] Nicholas M Luscombe, Dov Greenbaum, and Mark Gerstein. What is bioinformatics? an introduction and overview. *Yearbook of Medical Informatics*, 1:83–99, 2001.

[3] Francis H Crick. On protein synthesis. In *Symposia of the Society for Experimental Biology*, volume 12, page 138, 1958.

[4] U.S. National Library of Medicine National Center for Biotechnology Information. Genbank. `http://www.ncbi.nlm.nih.gov/genbank`. Consultada: 2013-10-02.

[5] Jack Dongarra, Thomas Sterling, Horst Simon, and Erich Strohmaier. High-performance computing: clusters, constellations, mpps, and future directions. *Computing in Science & Engineering*, 7(2):51–59, 2005.

[6] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1:1–40, 1993.

[7] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[8] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[9] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[10] Kang Zhang. *Visual languages and applications*. Springer, 2007.

[11] Nan C. Shu. *Visual programming*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[12] Sergio Víctor Chapa Vergara. *Programación automática a partir de descriptores de flujo de información*. PhD thesis, CINVESTAV-IPN, 1991.

[13] Adriana Hernandez Montoya. Lida/rec lenguaje visual para bases de datos. Master's thesis, CINVESTAV-IPN, 2005.

[14] Joaquín Sergio Zepeda Hernández. Modelo y arquitectura para exploración meta-conceptual dinámica en bases de datos con información biológica. Master's thesis, CINVESTAV-IPN, 2009.

[15] World Wide Web Consortium. Html 4.01 specification. `http://www.w3.org/TR/html401/`. Consultada: 2013-02-21.

[16] World Wide Web Consortium. Html 5.1 nightly. `http://www.w3.org/html/wg/drafts/html/master/`. Consultada: 2013-02-21.

[17] D. Crockford. The application/json media type for javascript object notation (json). `http://tools.ietf.org/html/rfc4627`. Consultada: 2013-02-21.

[18] D. Crocker. Augmented bnf for syntax specifications: Abnf. `http://tools.ietf.org/html/rfc4234`. Consultada: 2013-02-21.

[19] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Tim Carver, Matthew R. Pocock, and Anil Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:2004, 2004.

[20] The University of Manchester and the European Bioinformatics Institute. Biocatalogue. `https://www.biocatalogue.org/`. Consultada: 2013-10-02.

[21] The University of Manchester. Taverna frequently asked questions - does taverna provide services to use in workflows? `http://www.taverna.org.uk/documentation/faq/general/services-in-workflows/`. Consultada: 2013-10-02.

[22] The University of Manchester. Taverna frequently asked questions - how do i add a service to the service panel? `http://www.taverna.org.uk/documentation/faq/building-workflows/adding-a-service/`. Consultada: 2013-10-02.

[23] The University of Manchester. Mygrid developer wiki - creating a tool description. `http://dev.mygrid.org.uk/wiki/display/taverna/Creating+a+tool+description`. Consultada: 2013-10-02.

[24] University of Virginia. Tavernapbs. `http://cphg.virginia.edu/mackey/projects/sequencing-pipelines/tavernapbs/`. Consultada: 2013-10-02.

[25] The University of Manchester. Mygrid developer wiki - beanshell. `http://dev.mygrid.org.uk/wiki/display/taverna/Beanshell`. Consultada: 2013-10-02.

[26] University of Virginia. Creating a simple tavernapbs workflow. `http://cphg.virginia.edu/mackey/projects/sequencing-pipelines/tavernapbs/creating-a-simple-tavernapbs-workflow/`. Consultada: 2013-10-02.

[27] National Science Foundation. Kepler. `https://kepler-project.org/`. Consultada: 2013-10-02.

[28] Jeremy Goecks, Anton Nekrutenko, James Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, 2010.

[29] Galaxy Project. Galaxy main. `https://main.g2.bx.psu.edu/`. Consultada: 2013-10-02.

[30] Galaxy Project. Galaxy wiki - adding tools to galaxy. `http://wiki.galaxyproject.org/Admin/Tools/Add%20Tool%20Tutorial`. Consultada: 2013-10-02.

[31] Galaxy Project. Galaxy wiki - galaxy tool xml file. `http://wiki.galaxyproject.org/Admin/Tools/ToolConfigSyntax`. Consultada: 2013-10-02.

[32] Scott Walker, Alan Dearle, Graham Kirby, and Stuart Norcross. Exposing application components as web services. *arXiv preprint arXiv:1006.4504*, 2010.

[33] Ian Foster. The globus toolkit for grid computing. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 2–2. IEEE, 2001.

[34] Gopi Kandaswamy and Dennis Gannon. A mechanism for creating scientific application services on-demand from workflows. In *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 8–pp. IEEE, 2006.

[35] Oracle Corporation. Jersey - restful web services in java. `https://jersey.java.net/`. Consultada: 2013-10-02.