UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

MANUAL PARA LA CONFIGURACIÓN DE UN AMBIENTE DE DESARROLLO CON RUBY ON RAILS

ACTIVIDAD DE APOYO A LA DOCENCIA

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

JESÚS VILA SÁNCHEZ

DIRECTOR DE TESIS:

M. EN C. MARÍA GUADALUPE ELENA IBARGÜENGOITIA GONZÁLEZ

2018

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

1. Datos del alumno.
   Vila
   Sánchez
   Jesús
   5512594326
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   309280970

2. Datos del asesor.
   M. en C.
   Ibargüengoitia
   González
   María Guadalupe Elena

3. Datos del sinodal 1
   Dra.
   Oktaba
   Jadwiga
   Hanna

4. Datos del sinodal 2
   Dr.
   Valdés
   Souto
   Francisco

5. Datos del sinodal 3
   Dra.
   Martínez
   Ramírez
   Selene Marisol

6. Datos del sinodal 4
   M. en I.
   Ramírez
   Pulido
   Karla

7. Datos de la Tesis
   Manual para la configuración de un ambiente de desarrollo con Ruby on Rails
   81 p.
   2018

Índice de Códigos 5

Índice de Figuras 9

Índice de Tablas 10

Introducción 11

Capítulo 1. Antecedentes 13
1.1 Metodologías de trabajo 14
1.1.1 SCRUM 14
1.1.2 Kanban 15
Trello 15
1.1.3 Forma de trabajo 15
1.2. Planeación del sistema 15
1.2.1 Objetivos 16
1.2.2 Product backlog 17
Primera Iteración 18
Segunda Iteración 18
Tercera Iteración 18
Cuarta Iteración 18
Quinta Iteración 19

Capítulo 2. Conociendo Ruby on Rails 20
2.1 Ruby 20
2.1.1 Instalación 20
2.1.2 Gemas 20
2.1.2.1 Instalar una gema 21
2.1.2.2 Instalar gema con versión específica 21
2.1.2.3 Desinstalar una gema 21
2.1.2.4 Desinstalar gema por versión 21
2.1.2.5 Enlistar gemas 22
2.2 Ruby on Rails 22
2.2.1 Arquitectura Modelo-Vista-Controlador (MVC) 22
2.2.2 Instalación de Ruby on Rails 23
2.2.3 Creación de proyecto 24
2.2.3.1 Aplicación 25
2.2.3.2 Carpeta de configuraciones 25
2.2.3.3 Carpeta de Base de Datos 26
2.2.3.4 Gemfile 26
2.2.4 Variables de entorno 28
2.2.5 Configuración de credenciales de base de datos 29

2.2.6 Modelo 30
2.2.6.1 Migraciones 30
2.2.6.2 Validaciones 32
2.2.6.3 Operaciones 32
2.2.6.3.1 Crear 32
2.2.6.3.2 Leer 33
2.2.6.3.3 Actualizar 33
2.2.6.3.4 Eliminar 34
2.2.6.4 Relaciones de modelos 34
2.2.7 Controlador 35
2.2.7.1 Funciones de controladores 36
2.2.7.2 Rutas 37
2.2.8 Scaffold 38
Capítulo 3. Ambiente de desarrollo **39**
3.1 Descripción 39
3.1.1 Controlador de versiones y Alojamiento de Repositorio 40
3.1.1.1 Comandos básicos de Git 40
3.1.1.1.1 Configurar Git 40
3.1.1.1.2 Iniciar repositorio local 41
3.1.1.1.3 Estado del repositorio 41
3.1.1.1.4 Manejo de ramas 41
3.1.1.1.4.1 Crear rama 41
3.1.1.1.5 Guardar cambios 42
3.1.1.1.6 Subir cambios a repositorio remoto 43
3.1.1.1.7 Bajar cambios de repositorio remoto 43
3.1.1.1.8 Esconder cambios 43
3.1.1.1.9 Enlistar cambios escondidos 44
3.1.1.1.10 Mirar cambios escondidos 44
3.1.1.1.11 Recuperar cambios escondidos 44
3.1.2 Docker 45
3.1.2.1 Generar imagen 46
3.1.2.2 Enlistar imágenes 46
3.1.2.3 Remover imagen 47
3.1.2.4 Crear un contenedor 47
3.1.2.5 Enlistar contenedores 48
3.1.2.6 Detener un contenedor 48
3.1.2.7 Remover un contenedor 48
3.1.2.8 Comandos dentro del contenedor 49
3.1.2.9 Volúmenes de información 50

3.1.3 Dockerfile 50
3.1.4 Docker Compose 51
3.1.4.1 Construir/actualizar imágenes 54
3.1.4.2 Crear contenedores 55
3.1.4.3 Detener contenedores 56
3.1.4.4 Registros del contenedor 56
3.1.5 Heroku 57
3.1.5.1 Registros de la aplicación 59
3.1.5.2 Variables de entorno 59
3.1.5.3 Actualizar 60
3.1.5.4 Configuración de Base de Datos en Heroku 60
3.2 Desplegar en producción 62
3.2.1 Heroku con Docker 62
Capítulo 4. Ejemplos de desarrollo **64**
4.1 Manejo de sesión con Devise 64
4.1.1 Registro de usuario 67
4.2 Creación de recursos de manera asíncrona 68
4.2.1 Recursos con sesión obligatoria 68
4.3 Generación de archivos PDF 75
Conclusiones **79**
Bibliografía **80**

Índice de Códigos

Código 2.1.1.1 Llaves GPG 20
Código 2.1.1.2 Descarga a instalación de RVM 20
Código 2.1.2.1.1 Instalar una gema 21
Código 2.1.2.2.1 Instalar una gema con versión específica 21
Código 2.1.2.3.1 Desinstalar una gema 21
Código 2.1.2.4.1 Desinstalar una gema por versión 21
Código 2.1.2.5.1 Enlistar gemas 22
Código 2.2.2.1 Instalación de Ruby on Rails 23
Código 2.2.3.1 Creación de proyecto 24
Código 2.2.3.2 Ejemplo creación de proyecto 24
Código 2.2.3.4.1 Archivo Gemfile con manejo de grupos 27
Código 2.2.3.4.2 Instalar gemas que estén en archivo Gemfile 27
Código 2.2.3.4.3 Instalar gemas evitando grupos 27
Código 2.2.3.4.4 Instalar gemas evitando grupos production y test 28
Código 2.2.4.1 Archivo de configuración de variables de entorno 28
Código 2.2.4.2 Ejemplo de uso de varibles de entorno 29
Código 2.2.5.1 Configuración de credenciales de base de datos 29
Código 2.2.5.2 Comando para generar base de datos 30
Código 2.2.6.1 Comando general para crear modelos 30
Código 2.2.6.2 Ejemplo de uso de comando para generar modelos 30
Código 2.2.6.1.1 Archivo de configuración para migración de cursos 31
Código 2.2.6.1.2 Agregar migración 31
Código 2.2.6.1.3 Aplicar cambios de migraciones 31
Código 2.2.6.1.4 Cargar esquema de base de datos 32
Código 2.2.6.2.1 Validación de presencia de un campo antes de almacenar 32
Código 2.2.6.3.1.1 Crear y almacenar objetos 33
Código 2.2.6.3.3.1 Actualizar registros de la base de datos 34
Código 2.2.6.3.4.1 Eliminar registros de la base de datos 34
Código 2.2.6.4.1 Obtener cursos de profesor sin manejo de relaciones entre modelos 35
Código 2.2.6.4.2 Manejo de relaciones entre modelos y obtención de cursos 35
Código 2.2.7.1 Iniciar controlador de cursos 35
Código 2.2.7.1.1 Firmas de las funciones dentro de controlador 37
Código 2.2.7.2.1 Archivo de configuración de rutas 38
Código 2.2.8.1 Comando scaffold 38
Código 3.1.1.1.1.1 Configuración de nombre de usuario 40

Código 3.1.1.1.1.2 Configuración de correo electrónico 41
Código 3.1.1.1.2.1 Iniciar repositorio local 41
Código 3.1.1.1.3.1 Obtener estado del repositorio 41
Código 3.1.1.1.4.1 Creación de rama 42
Código 3.1.1.1.4.2 Creación y cambio de rama 42
Código 3.1.1.1.5.1 Actualización Git archivo por archivo 42
Código 3.1.1.1.5.2 Actualización Git todos los archivos de la carpeta 42
Código 3.1.1.1.5.3 Confirmar cambios (compacta) 42
Código 3.1.1.1.5.4 Confirmar cambios (extendida) 43
Código 3.1.1.1.6.1 Subir cambios a repositorio remoto con paŕametros 43
Código 3.1.1.1.6.2 Subir cambios a repositorio remoto sin paŕametros 43
Código 3.1.1.1.7.1 Bajar cambios de rama master del repositorio remoto origin 43
Código 3.1.1.1.8.1 Esconder cambios repositorio local 44
Código 3.1.1.1.9.1 Enlistar cambios escondidos 44
Código 3.1.1.1.10.1 Mostrar archivos modificado en cambios escondidos 44
Código 3.1.1.1.10.1 Mostrar archivos modificado en cambios escondidos índice 2 44
Código 3.1.1.1.11.1 Recuperar último cambio escondido 44
Código 3.1.1.1.11.2 Recuperar cambio escondido de índice 1 45
Código 3.1.2.1 Demonio dockerd 45
Código 3.1.2.2 Iniciar dokcerd desde línea de comandos 45
Código 3.1.2.2 Iniciar dokcerd desde línea de comandos con privilegios 46
Código 3.1.2.3 Obtener información acerca del comando docker 46
Código 3.1.2.1.1 Generar y nombrar imagen de Docker 46
Código 3.1.2.1.2 Generar, nombrar y versionar imagen de Docker 46
Código 3.1.2.1.3 Mostrar opciones para generar una imagen de Docker 46
Código 3.1.2.2.1 Mostrar imágenes 47
Código 3.1.2.2.2 Mostrar todas las imágenes de Docker 47
Código 3.1.2.3.1 Remover imagen en base a su identificador 47
Código 3.1.2.3.2 Remover todas las imágenes inactivas 47
Código 3.1.2.4.1 Crear contenedor y asociar puertos, ejecución en segundo plano 47
Código 3.1.2.5.1 Obtener todos los contenedores activos 48
Código 3.1.2.6.1 Detener un contenedor 48
Código 3.1.2.6.2 Detener ejecución de todos los contenedores 48
Código 3.1.2.7.1 Remover contenedores específicos 49
Código 3.1.2.7.2 Remover todos los contenedores 49
Código 3.1.2.8.1 Comando dentro de contenedor con flujo de datos abierto 49
Código 3.1.2.8.2 Ejemplo de comando dentro de contenedor con flujo de datos abierto 49
Código 3.1.2.8.3 Ejecutar comando dentro de un contenedor sin flujo de datos 49
Código 3.1.2.9.1 Forma general de creación de volumen en Docker 50

Código 3.1.2.9.2 Ejemplo de creación de volumen en Docker 50
Código 3.1.3.1 Archivo de configuración Dockerfile 51
Código 3.1.4.1 Archivo docker-compose.yml que inicia un contenedor de PostgreSQL 52
Código 3.1.4.2 Archivo docker-compose.yml que inicia un contenedor de PostgreSQL con volúmenes de información y redes 53
Código 3.1.4.3 Archivo docker-compose.yml que inicia un contenedor de la aplicación y se conecta con un contenedor de PostgreSQL con volúmenes de información y redes 54
Código 3.1.4.1.1 Generar/actualizar imagen de todos los servicios 55
Código 3.1.4.1.2 Generar/actualizar imagen de un servicio 55
Código 3.1.4.1.3 Generar imágenes a partir de archivo en especifico 55
Código 3.1.4.2.1 Crear contenedores con comando docker-compose 55
Código 3.1.4.2.2 Crear contenedores y mantener en segundo plano 55
Código 3.1.4.3.1 Detener contenedores asociados al archivo docker-compose.yml 56
Código 3.1.4.3.2 Detener contenedores y volúmenes asociados al archivo docker-compose.yml 56
Código 3.1.4.4.1 Comando para obtener todos los registros de un archivo docker-compose.yml 57
Código 3.1.4.4.2 Obtener registros del servicio db 57
Código 3.1.4.4.3 Obtener las últimas 10 entradas de los registros y mantener flujo 57
Código 3.1.5.1 Iniciar sesión en interfaz de línea de comandos de Heroku 58
Código 3.1.5.2 Iniciar aplicación en Heroku 58
Código 3.1.5.3 Configuración de archivo Procfile 59
Código 3.1.5.1.1 Registros de la aplicación en Heroku 59
Código 3.1.5.2.1 Configuración de variables de entorno manual 59
Código 3.1.5.2.2 Configuración de variables de entorno mediante figaro 60
Código 3.1.5.2.3 Obtener lista de variables de entorno definidas 60
Código 3.1.5.3.1 Actualización de cambios desde rama master local 60
Código 3.1.5.3.2 Actualización de cambios desde otra rama local 60
Código 3.1.5.4.1 Complemento para base de datos Heroku Postgres con CLI 61
Código 3.2.1.1 Iniciar sesión en el Registro de Contenedores de Heroku 62
Código 3.2.1.2 Archivo Dockerfile para Heroku 63
Código 3.2.1.3 Comando para crear contenedor en Heroku 63
Código 4.1.1 Comando para iniciar configuración básica de gema Devise 64
Código 4.1.2 Configuración de dirección web del servidor anfitrión 65
Código 4.1.4 Asociación de gema Devise con un modelo 65
Código 4.1.5 Asociación de gema Devise con model User 65
Código 4.1.6 Activar gema Devise en más de un modelo 66
Código 4.1.7 Comando para generar vistas/controladores de módulos 66
Código 4.1.8 Comando para generar todas las vistas asociadas a User 66
Código 4.1.9 Comando para generar ciertas vistas asociadas a User 66

Código 4.1.10 Comando para generar ciertos controladores asociados a User 66
Código 4.2.1.1 Archivo con la migración de los profesores 69
Código 4.2.1.2 Archivo con la migración de los cursos 70
Código 4.2.1.3 Configuración de modelo de profesores 70
Código 4.2.1.4 Configuración de modelo de cursos 71
Código 4.2.1.5 Configuración de controlador de cursos 75
Código 4.3.1 Inicializar gema WickedPDF 75
Código 4.3.2 Agregar PDF a tipo de respuesta 75
Código 4.3.3 Plantilla HTML de comentarios de usuario 76
Código 4.3.4 Código para responder con archivo PDF 77
Código 4.3.5 Código para que se descargue archivo PDF 78

Índice de Figuras

Figura 1.2.1.1. Diagrama General de casos de uso 16
Figura 2.2.1.1. Comunicación en arquitectura MVC [5] 23
Figura 2.2.1.2. Diagrama de distribución [5] 23
Figura 2.2.6.1 Información de salida del comando para generar modelos 30
Figura 2.2.7.1 Salida de comando para generar controladores 36
Figura 3.1.5.4.1 Sección de recursos de una aplicación en Heroku 60
Figura 3.1.5.4.2 Complemento para base de datos Heroku Postgres 61
Figura 3.1.5.4.3 Versiones del complemento Heroku Postgres 61
Figura 4.1.1.1 Registro predeterminado de Devise 67
Figura 4.1.1.2 Registro modificado de Devise con estilos 67
Figura 4.2.1.1 Diagrama Entidad-Relación de profesores y cursos 69
Figura 4.3.1 PDF generado a partir de plantilla HTML 78

Índice de Tablas

Tabla 1.2.2.1 Product backlog 18
Tabla 3.1.1 Ambiente de desarrollo 40

Introducción

En este manual se expone cómo es el proceso, desde una vista técnica con *Ruby on Rails*, relacionado con la planeación, desarrollo y despliegue de un sistema web para un cliente real[1].

Este trabajo abarca conceptos generales de *Ruby on Rails*, para que los alumnos de cursos (como Ingeniería de Software I y II, Administración de Empresas de Software, etcétera) que deseen usar *Ruby on Rails* se familiaricen con los conceptos que se expondrán a lo largo del manual. Se presenta una explicación acerca de cómo se genera la estructura del código, porqué se utilizaron ciertas bibliotecas, qué ventajas y desventajas ofrecían respecto a tiempo y dificultad; la arquitectura que se utilizó, el sistema manejador de bases de datos y las configuraciones que se usaron, tratando que el alumno tenga una noción más amplia de lo que implica el desarrollo de un sistema en esta plataforma y no se limite solamente a la implementación de código.

Se explica (brevemente) cuáles fueron las metodologías que se usaron durante el desarrollo del sistema, qué herramientas se utilizaron para dar seguimiento a las tareas y cómo fue el desarrollo del sistema que abarcó dos etapas, el curso de Ingeniería de Software II y el servicio social.

La Secretaría de Educación Abierta y Continua (SEAyC) de la Facultad de Ciencias fungió como el cliente del proyecto, esta Secretaría tiene la función de promover y organizar cursos afines a las áreas de conocimiento de la Facultad (aunque no sólo se remite a éstas). El sistema que el cliente solicitó debía cumplir con:

- Agilizar el proceso de captura de formatos tanto de los curriculums de los profesores interesados en impartir curso, como de los cursos que se quieren impartir.
- Permitir el registro de estudiantes a la plataforma para que se les puedan informar cuando un curso les interesa y así poder dar un seguimiento a los cursos que atraen a las personas.

[1] Como plataforma de desarrollo se da un enfoque agnóstico en algunas secciones.

- Procesar la información proporcionada por los profesores para que se generen los archivos que la secretaría hace llegar al Comité Académico (CA) para que éste realice las observaciones pertinentes.

Se exponen algunas herramientas que se recomiendan cuando se realicen trabajos en equipo que no sólo se limitan al desarrollo de un sistema web, así como algunas plataformas que proveen de servicios de alojamiento, manejo de base de datos, etcétera.

En el primer capítulo se presenta de una forma más detallada el planteamiento del problema así como se exponen algunas metodologías ágiles que se usaron, ya que esto se cubre más a fondo en el trabajo *Creación y Documentación de un Sistema Web para la Secretaría de Educación Abierta y Continua [5].*

En el segundo capítulo se cubren conceptos generales tanto de *Ruby* como de *Ruby on Rails,* cómo realizar la instalación, comandos básicos para el manejo de la herramienta, así como recomendaciones para ciertas configuraciones como las variables de entorno.

En el tercer capítulo se cubre cómo realizar la configuración de un ambiente de desarrollo para trabajo en equipo, que incluye cómo tener el código en un repositorio remoto, cómo configurar el proyecto con *Docker* para evitar problemas de versiones y cómo hacer el despliegue de la aplicación a una plataforma como *Heroku*.

En el cuarto capítulo se muestran algunos ejemplos de desarrollo que se hicieron durante el desarrollo del sistema de la Secretaría de Educación Abierta y Continua de la Facultad de Ciencias que pueden ser de utilidad al lector, tales como el manejo de sesión y generación de PDF de manera automática.

Capítulo 1. Antecedentes

En la SEAyC el proceso para registrar una propuesta de curso implica el uso de documentos impresos, estos documentos son el currículum vitae de los profesores y la propuesta del curso que debe seguir el formato que proporciona la SEAyC a través de su página. Cuando la SEAyC recibe la documentación necesaria hace llegar éstos al CA para que éste se encargue de realizar las observaciones pertinentes.

Una vez que la documentación ha sido aprobada por el CA, la SEAyC, solicita al Consejo Técnico de la Facultad su autorización para así poder realizar los trámites necesarios con el Departamento de Contabilidad, de manera que se puedan recibir los pagos de las personas que se vayan a inscribir al curso una vez que éste ya haya sido abierto. Por otro lado, la secretaría se encarga de dar difusión a los cursos en su página permitiendo a los alumnos mostrar interés en uno o más cursos afines a sus gustos, sin embargo el proceso para conocer el número de personas que están interesadas sigue siendo un proceso manual lo que conlleva que un curso tarde más en ser abierto puesto que no hay un sistema de notificaciones que informe al coordinador del área que el curso ha interesado a varias personas.

Con la intención de promover los cursos que la SEAyC ofrece al público así como acelerar el proceso para el registro de nuevas propuestas, se desarrollará un sistema capaz de cubrir estas funcionalidades. Si bien es cierto que la SEAyC ya cuenta con un sistema, éste no cubre los problemas aquí planteados.

Mediante juntas periódicas entre algunos miembros pertenecientes a la SEAyC y el equipo de desarrollo, se lograron establecer los requerimientos que se consideraban prioritarios y que podían lograrse dentro del tiempo disponible.

1.1 Metodologías de trabajo

1.1.1 SCRUM

Es un marco de trabajo para el desarrollo de productos complejos desarrollado por Ken Schwaber y Jeff Sutherland a principios de los años 90, en éste se emplea un enfoque iterativo e incremental y toman decisiones con base en lo que se conoce.[1]

Consta de tres interesados, el dueño del producto, el equipo de desarrollo y el scrum master [5]:

- El dueño del producto es el responsable de gestionar el *“Product Backlog”* y define el objetivo y alcance del producto.
- El equipo de desarrollo, es el responsable de desarrollar el producto, el tamaño del equipo no debe de ser ni muy grande ni muy pequeño, debe ser de un tamaño que permita completar cantidades de trabajo significativas pero que permita que la complejidad de la organización sea la suficiente para poder gestionarse por un proceso empírico;
- El Scrum master es el responsable de que todos los involucrados sepan qué es Scrum y que lo entiendan para poder adoptarlo en el desarrollo.

Se maneja por iteraciones que tienen una duración aproximada de un mes y consiste de varias prácticas como lo son [5]:

- La planeación de la iteración, definir el objetivo de la iteración
- Reuniones diarias de no más de 15 minutos en donde los miembros del equipo de desarrollo exponen lo que han hecho desde la última reunión y hacen una proyección de lo que se hará hasta la siguiente reunión
- Retrospectiva de la iteración, en donde se identifican los elementos más importantes que salieron bien y las posibles mejoras, así como un plan para implementar las mejoras en las próximas iteraciones.

Una vez que se ha empezado la iteración no se pueden hacer cambios que afecten su objetivo.

1.1.2 Kanban

Es una metodología de trabajo que permite llevar la administración del trabajo, en el que se representan las tareas de un flujo de trabajo mediante tarjetas y un tablero de seguimiento donde se puede seguir el estado de las tareas, que pueden estar en alguno de los tres estados que se manejan principalmente, que son: *Por Hacer (To Do)*, *Haciendo (Doing)*, *Hecho (Done).* Este tablero debe ser visible para todos los miembros del equipo, a partir de éste se podrá saber qué están haciendo los demás miembros, si se están cumpliendo los tiempos estimados y detectar problemas con la planeación.

Ya que las tareas no sólo se remiten al ámbito de desarrollo se recomienda usar tarjetas de diferentes colores que permitan distinguir los diferentes tipos de trabajo, dígase administrativo, de documentación, pruebas, etcétera. [2]

Trello

Herramienta en línea que está inspirada en la metodología Kanban, permite visualizar el flujo de trabajo, utilizar colores diferentes para las actividades que se realizan, así como dar seguimiento al tiempo utilizado por cada tarjeta. [5]

1.1.3 Forma de trabajo

Una vez que se tiene la lista de requerimientos se escribe el plan de trabajo y se definen las iteraciones haciendo uso del “*Product backlog*”.

Durante las reuniones de equipo que se daban antes de iniciar cada iteración se decidía qué requerimientos iba a realizar cada integrante del equipo y se les daba seguimiento mediante un tablero en la plataforma Trello.

Al final de cada iteración, como equipo, verificamos que todos los requerimientos desarrollados funcionaran como era debido, que se hubiera desarrollado todo lo que se tenía planeado para esa iteración y se escribía la retrospectiva de la iteración en la que hacíamos notar que cosas habíamos realizado bien, cosas que se podían mejorar para futuras iteraciones y equivocaciones que habíamos cometido y que no nos podíamos permitir que volvieran a suceder. También se buscaba agendar una reunión con los miembros de la SEAyC para mostrar los avances que se realizaron durante la iteración, de este modo ellos nos podían hacer observaciones tanto estéticas como técnicas del sistema lo que nos permitía mejorar el sistema en las futuras iteraciones.

El proyecto se montaba sobre la plataforma de *Heroku* para que, tanto los integrantes del equipo de desarrollo como los miembros de la SEAyC, tuvieran acceso al sistema y así poder realizar pruebas y familiarizarse.

El desarrollo del sistema se hizo en dos etapas, una que fue durante el curso de Ingeniería de Software II y otra durante el Servicio Social.

1.2. Planeación del sistema

En la asignatura de Ingeniería de Software II, durante el semestre 2016-1, se inició la planeación y el desarrollo del sistema. Esta etapa tenía la finalidad de promover el trabajo en equipo y acercarnos a un ambiente laboral.

1.2.1 Objetivos

Implementar un sistema web que permita agilizar los registros de los ponentes así como las propuestas de cursos, registrar a las personas interesadas, llevar un registro de las personas interesadas por curso, promover los cursos dentro de la página, poder exportar los archivos que se tienen que presentar antes las diferentes instancias participantes (Comité Académico, Consejo Técnico, etc), enviar notificaciones a las personas interesadas cuando se haya abierto el curso, enviar notificaciones a los ponentes y personal de la SEAyC cuando un curso haya alcanzado el número mínimo de participantes estipulado en el curso. Ver figura 1.2.1.1.

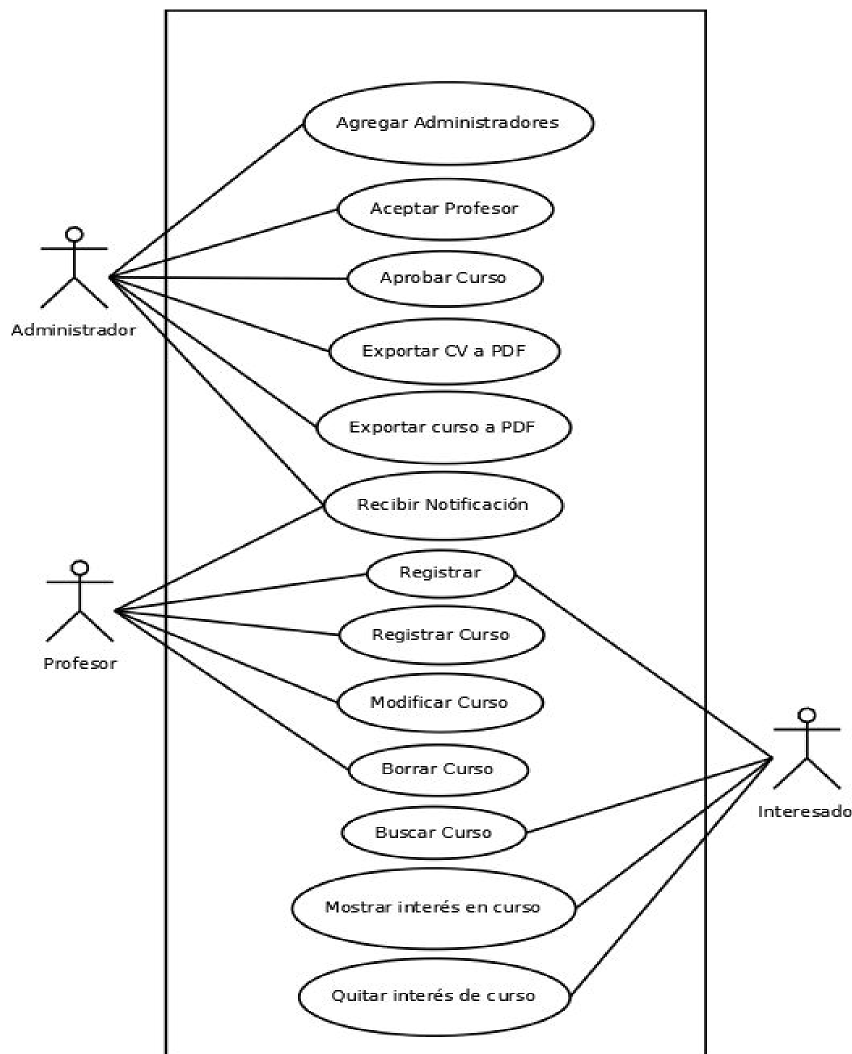

Figura 1.2.1.1. Diagrama General de casos de uso

1.2.2 Product backlog

Después de tener una reunión con los miembros de la SEAyC y de plantear el problema se identificaron los siguientes requerimientos así como la prioridad de cada uno.

Id	Funcionalidad	Prioridad	Estimación
01	Registrar CV de los ponentes	Alta	6hrs.
02	Modificar CV de los ponentes	Media	2hrs.
03	Registrar propuesta de curso	Alta	6hrs.
04	Revisar CV de los ponentes	Baja	2hrs.
05	Revisar propuesta de curso	Media	4hrs.
06	Borrar curso	Media	1hrs.
07	Registrar al coordinador	Baja	4hrs.
08	Autorizar Ponente	Alta	4hrs.
09	Autorizar propuesta de curso	Alta	3hrs.
10	Clasificar el catálogo de cursos	Media	4hrs.
11	Registrar interesados	Media	5hrs.
12	Interesarse en un curso	Baja	4hrs.
13	Enviar notificación	Alta	2hrs.
14	Realizar un reporte de los interesados	Media	5hrs.
15	Registrar los números de interesados	Alta	3hrs.

Tabla 1.2.2.1 Product backlog

Los tiempos que se estimaron para los requerimientos no siempre se cumplieron ya que para las primeras iteraciones se tardó un poco más del tiempo estimado, puesto que los integrantes del equipo estaban en el proceso de aprendizaje de las herramientas que se utilizaron durante el desarrollo del sistema. Una vez que los integrantes tuvieron el conocimiento necesario los tiempos en los que se terminaban los requerimientos eran más cercanos a los que se habían estimado en un principio aunque aún habían retardos por falta de experiencia al hacer las estimaciones.

La iteración en la que se realizaban las funcionalidades dependía de la prioridad de éstas para el cliente, pero no siempre se podía desarrollar una que fuera de mayor prioridad ya que podía depender de otras que no estaban consideradas de prioridad alta. Por lo que en un principio se decidió que las iteraciones quedarían conformadas de la siguiente manera.

Primera Iteración

- Registrar CV de los ponentes
- Modificar CV de los ponentes
- Registrar propuesta de curso

Segunda Iteración

- Revisar CV de los ponentes
- Revisar propuesta de curso
- Borrar curso
- Corrección de errores

Tercera Iteración

- Registrar coordinador
- Autorizar ponente
- Autorizar propuesta de curso
- Corrección de errores

Cuarta Iteración

- Clasificar el catálogo de cursos
- Registrar interesados
- Interesarse en curso
- Corrección de errores

Quinta Iteración

- Enviar notificación
- Realizar reporte de interesados
- Registrar número de interesados
- Corrección de errores

En este capítulo se describieron los antecedentes del proyecto, qué metodologías se usaron, la forma de trabajo, cómo fue el proceso de planeación, cuáles fueron los requerimientos del proyecto, los tiempos que se estimaron y las iteraciones que se manejaron durante el desarrollo del proyecto.

Capítulo 2. Conociendo *Ruby on Rails*

2.1 *Ruby*

Para el desarrollo del sistema se utilizó el lenguaje de programación *Ruby* ya que tiene una sintaxis muy simple. Aunado a que tiene una gran comunidad que ayuda a resolver los problemas y una gran cantidad de bibliotecas externas listas para integrarse al proyecto.

2.1.1 Instalación

Para instalar *Ruby* se recomienda usar el Gestor de Versiones de *Ruby* (*RVM*, *Ruby Version Manager*), ya que permite tener varias versiones del mismo sin generar conflictos.

Primero se deben de agregar las llaves GPG (GNU Privacy Guard) que se usan para garantizar la seguridad de la instalación, como se puede ver en el Código 2.1.1.1.

```
$ gpg --keyserver hkp://keys.gnupg.net --recv-keys
409B6B1796C275462A1703113804BB82D39DC0E3
7D2BAF1CF37B13E2069D6956105BD0E739499BDB
```

Código 2.1.1.1 Llaves *GPG*

Una vez agregadas las llaves ya podemos instalar RVM en su versión estable (recomendada), ver Código 2.1.1.2, si se desea instalar la versión en desarrollo basta con remover la bandera *-s stable.*

```
$ \curl -sSL https://get.rvm.io | bash -s stable
```

Código 2.1.1.2 Descarga a instalación de *RVM*

2.1.2 Gemas

Las gemas son bibliotecas y programas de *Ruby* distribuidos mediante el manejador de paquetes *RubyGems*, este programa ya viene instalado con *Ruby*. Se recomienda hacer

uso de las gemas ya que (la mayoría) son funcionalidades listas para su uso las cuales ayudan a minimizar los tiempos de desarrollo y la complejidad del proyecto.

Los comandos más básicos y de los que más se utilizan para el manejo de las gemas son:

2.1.2.1 Instalar una gema

```
$ gem install nombre_gema
```

Código 2.1.2.1.1 Instalar una gema

2.1.2.2 Instalar gema con versión específica

Para instalar una gema por versión se utiliza la bandera *-v* seguida por la versión que se quiere utilizar, esto es útil cuando quieres utilizar funciones de la gema que en versiones más recientes se vuelven obsoletas o han desaparecido.

```
$ gem install nombre_gema -v X.Y.Z
```

Código 2.1.2.2.1 Instalar una gema con versión específica

2.1.2.3 Desinstalar una gema

```
$ gem uninstall nombre_gema
```

Código 2.1.2.3.1 Desinstalar una gema

2.1.2.4 Desinstalar gema por versión

Hay veces en las que se tiene más de una versión instalada de la misma gema lo cual puede generar conflictos ya que se utiliza la versión más reciente, para desinstalar una versión específica al igual que en la instalación se pasa la bandera *-v* junto con la versión a desinstalar.

```
$ gem uninstall nombre_gema -v X.Y.Z
```

Código 2.1.2.4.1 Desinstalar una gema por versión

2.1.2.5 Enlistar gemas

Para saber cuáles gemas se tienen instaladas se tiene un comando que también dice las versiones que se tienen de cada gema.

```
$ gem list
```

Código 2.1.2.5.1 Enlistar gemas

2.2 Ruby on Rails

Ruby on Rails es un marco de trabajo para el desarrollo de proyectos web desarrollado por David Heinemeier Hansson liberando la versión 1.0 en Diciembre del 2005, maneja la arquitectura Modelo-Vista-Controlador (MVC) y principios de desarrollo de software que minimizan los tiempos de desarrollo como Convention over Configuration (CoC, *Convención sobre Configuración*) que genera una configuración básica sobre la cual se puede trabajar aunque también permite modificarlo según las necesidades; así como Don't Repeat Yourself (DRY, *No te Repitas*) que trata de evitar que se tenga código duplicado ya que dificulta el mantenimiento[2].

2.2.1 Arquitectura Modelo-Vista-Controlador (MVC)

Dada la necesidad de un sistema robusto para futuros cambios, se utilizó el patrón arquitectónico Modelo-Vista-Controlador que permite mantener el sistema en diferentes módulos promoviendo la menor dependencia entre cada módulo. [3]

El modelo es donde el sistema define los datos con los que va a trabajar, contiene funciones que permiten el acceso, creación, modificación y remoción de los datos. [3]

En la vista se desarrolla toda la parte visual del sistema, la interfaz de usuario y cómo se va a comunicar con el sistema. [3]

El controlador maneja todas las solicitudes hechas por la vista conectándose al modelo para obtener la información y regresar a la vista los datos obtenidos para que pueda informar al usuario el estado de la solicitud dependiendo de la respuesta. [3]

[2] https://guides.rubyonrails.org/getting_started.html#what-is-rails-questionmark

Una forma ver estos tres módulos es, el modelo como los objetos de la aplicación, la vista como toda la parte visual con la que el usuario puede interactuar y el controlador como la lógica que maneja la interacción del usuario con el sistema.

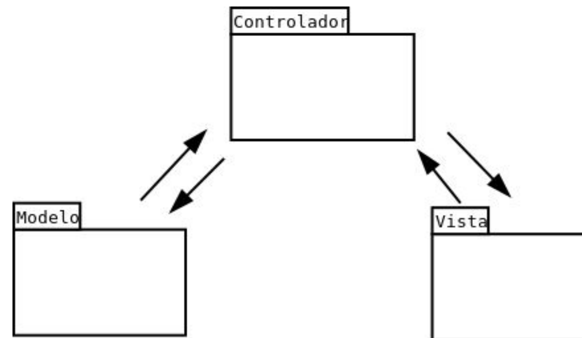

Figura 2.2.1.1 Comunicación en arquitectura MVC [5]

Este patrón arquitectónico distribuye los módulos en dos entidades, cliente y servidor. El cliente se encarga de mostrar la parte visual y realizar peticiones de información al servidor. El servidor se encarga de procesar las peticiones hechas por el cliente y dependiendo del tipo de solicitud se conecta con la base de datos para realizar alguna acción. [5]

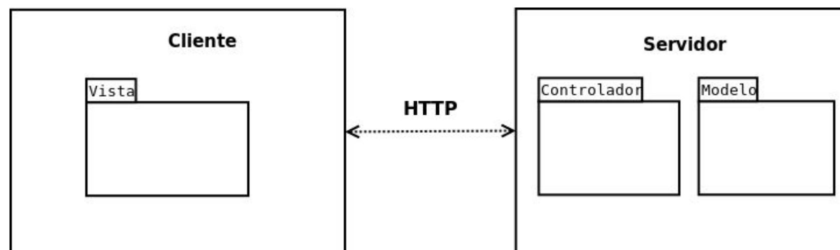

Figura 2.2.1.2 Diagrama de distribución [5]

2.2.2 Instalación de *Ruby on Rails*

Una vez se tiene instalada la versión de *Ruby* se ejecuta el siguiente comando para obtener la gema de *Ruby on Rails*, en este caso hacemos uso de la versión 4.2.3.

```
$ gem install rails -v 4.2.3
```

Código 2.2.2.1 Instalación de Ruby on Rails

Si la gema se instala de manera correcta ahora se puede utilizar el comando *rails* en la línea de comandos.

2.2.3 Creación de proyecto

Ya que se ha instalado el comando *rails*, se ejecuta de la siguiente manera para iniciar un proyecto:

```
$ rails new NOMBRE_DEL_PROYECTO --database=postgresql -T
```

Código 2.2.3.1 Creación de proyecto

```
$ rails new seayc --database=postgresql -T
```

Código 2.2.3.2 Ejemplo creación de proyecto

El cual genera una carpeta con la estructura básica de un proyecto de Ruby on Rails, la bandera *--database* se utiliza para indicar el sistema manejador de base de datos que vamos a utilizar, en este caso PostgreSQL, algunos otros manejadores disponibles son *mysql, oracle, sqlite, etc*. La bandera *-T* nos permite omitir todos los archivos de prueba del proyecto.

El proyecto creado maneja una estructura simple que separa por carpeta las partes del proyecto, las carpetas que más se van a modificar son *app/, config/, db/* y el archivo *Gemfile.*

- app/
- bin/
- config/
- config.ru
- db/
- Gemfile
- Gemfile.lock
- lib/
- log/
- public/
- Rakefile
- README.rdoc

- tmp/
- vendor/

2.2.3.1 Aplicación

La carpeta de la aplicación contiene los módulos modelo, vista, controlador. Un detalle de los modelos es que en éstos no se van a poner los campos de la tabla, en estos se realizan las relaciones con otros modelos que se tienen, así como validaciones necesarias de ejecutar antes de almacenar la instancia del modelo en la base de datos. Esta carpeta es donde va a estar la mayor parte de código, dentro de esta se encuentran los siguientes archivos.

- app/
  - assets/
  - controllers/
  - helpers/
  - mailers/
  - models/
  - views/

2.2.3.2 Carpeta de configuraciones

En esta carpeta se tienen las configuraciones del proyecto, como la configuración para conectarse a una base de datos, el manejo de las rutas del proyecto, configuración de variables de entorno, configuración de sistema de envío de correos, etcétera. Dentro de esta se carpeta se encuentran los siguientes archivos.

- config/
  - application.rb
  - application.yml
  - boot.rb
  - database.yml
  - environment.rb
  - environments/
  - initializers/
  - locales/
  - routes.rb

2.2.3.3 Carpeta de Base de Datos

En esta carpeta es donde se van a definir los campos de los modelos dentro de migraciones[3], generar un esquema general de la base de datos que se recomienda utilizar para generar la base de datos en otro equipo de cómputo, así como insertar valores predefinidos a la base de datos, por ejemplo, llenar un catálogo con los estados de la República Mexicana.

- db/
  - migrate/
  - schema.rb
  - seed.rb

2.2.3.4 Gemfile

En el archivo Gemfile se agregan las gemas necesarias para el desarrollo del sistema, como primer línea se tiene la ruta en donde se van a buscar los paquetes y después los nombres de las gemas a utilizar y en caso de ser necesario la versión requerida. Este archivo es muy útil ya que en caso de querer instalar el proyecto en otro sistema basta con ejecutar un comando e instala todas las dependencias necesarias según el entorno.

[3] Son archivos mediante los cuales se puede actualizar la base de datos sin tener que hacer uso de SQL

```
1  source 'https://rubygems.org' # Fuente de gemas
2  # Manejo de versiones de gemas
3  # Instala la versión más reciente de conector de PostgreSQL
4  gem 'pg'
5  # Instala la versión 4.2.3 de rails
6  gem 'rails', '4.2.3'
7  # Instala la versión 1.3.0 o una más reciente en caso de que exista
8  gem 'uglifier', '>= 1.3.0'
9  # Instala una versión mayor o igual a la 2.0 pero menor a la 3.0
10 gem 'jbuilder', '~> 2.0'
11 # Instala una versión mayor o igual a la 3.1.7 pero menor a la 3.2.0
12 gem 'bcrypt', '~> 3.1.7'
13 # También se pueden instalar gemas dependiendo del entorno
14 # Estas gemas sólo estarán disponibles cuando el proyecto esté en
15 # desarrollo o en pruebas
16 group :development, :test do
17  gem 'byebug'
18  gem 'web-console', '~> 2.0'
19 end
20 # Gemas para producción
21 group :production do
22  gem 'puma'
23 end
```

Código 2.2.3.4.1 Archivo *Gemfile* con manejo de grupos

Para instalar las gemas se utiliza el siguiente comando.

```
$ bundle install
```

Código 2.2.3.4.2 Instalar gemas que estén en archivo *Gemfile*

Si no se quieren instalar las gemas de algún grupo se puede usar la bandera *without*

```
$ bundle install --without grupo_1:grupo_2
```

Código 2.2.3.4.3 Instalar gemas evitando grupos

En el Código 2.2.3.4.4 se muestra cómo evitar instalar las gemas que se encuentren en los grupos *production* y *test*.

```
$ bundle install --without production:test
```

Código 2.2.3.4.4 Instalar gemas evitando grupos *production y test*

2.2.4 Variables de entorno

Se usan las variables de entorno para mantener secreta y segura cierta información[4] como contraseñas, cuentas de servicios de terceros, correo electrónico, teléfono, etcétera.

Para manejar las variables de entorno se utilizó la gema Figaro, la cual nos genera un archivo (*config/application.yml*) en donde se escriben todas las variables que queremos usar. Una ventaja de usar esta gema es que se encarga de realizar las configuraciones para leer el archivo, también permite agregar variables por ambiente y así agregar ciertos valores para desarrollo y otros para producción, como la dirección del servidor, la cuenta de correo, etcétera.

Para utilizar la gema basta agregarla al archivo Gemfile y actualizar las dependencias mediante el comando bundle (ver Código 2.2.3.4.2).

Para definir una variable se tiene que poner el nombre de la variable y su valor en el archivo *config/application.yml* que la gema Figaro generó. Por ejemplo, en las líneas 2 y 3 del código 2.2.4.1 se puede ver como se definen las variables de entorno *sendgrid_email y sendgrid_password*, cuyos valores sólo se necesitan cambiar en este archivo.

```
1. # config/application.yml
2. sendgrid_email = 'correo@secreto.com'
3. sendgrid_password = 'contraseña_secreto'
```

Código 2.2.4.1 Archivo de configuración de variables de entorno

Para utilizar las variables tan sólo se necesita utilizar una variable de tipo hash llamada ENV con la cual se pueden obtener los valores definidos, como se puede observar en las líneas n+4 y n+5 del Código 2.2.4.2.

[4] https://github.com/laserlemon/figaro

```
1. # config/environments/development.rb
2. Rails.application.configure do
3.   # Settings specified here will take precedence over those in config/application.rb.
n.   ...
n+1.  config.action_mailer.delivery_method = :smtp
n+2.    config.action_mailer.smtp_settings = {
n+3.      enable_starttls_auto: true,
n+4.      user_name: ENV['sendgrid_email'],
n+5.      password: ENV['sendgrid_password']
n+6.    }
n+7. end
```

Código 2.2.4.2 Ejemplo de uso de varibles de entorno

2.2.5 Configuración de credenciales de base de datos

Para configurar las credenciales de acceso a la base de datos se debe modificar el archivo *config/database.yml* (ver Código 2.2.5.1), el cual se crea con la configuración básica. Se recomienda poner toda la información sensible dentro de las variables de ambiente para que no sean compartidas dentro del repositorio.

```
# config/database.yml
development:
  <<: *default
  database: <%= ENV['NOMBRE_BASE_DE_DATOS_DESARROLLO'] %>
  username: <%= ENV['USARIO_BASE_DE_DATOS_DESARROLLO'] %>
  password: <%= ENV['CONTRASENA_BASE_DE_DATOS_DESARROLLO'] %>
  # host: DIRECCIÓN_DE_HOST     # default: localhost
  # port: PUERTO_DE_CONEXIÓN    # default: 5432 para PostgreSQL

production:
  <<: *default
  database: <%= ENV['NOMBRE_BASE_DE_DATOS_PRODUCCIÓN'] %>
  username: <%= ENV['USARIO_BASE_DE_DATOS_PRODUCCIÓN'] %>
  password: <%= ENV['CONTRASENA_BASE_DE_DATOS_PRODUCCIÓN'] %>
```

Código 2.2.5.1 Configuración de credenciales de base de datos

Ya que se configuraron los valores para la base de datos se necesita ejecutar el siguiente comando (ver Código 2.2.5.2) para generar la base de datos en nuestro sistema manejador de base de datos.

```
$ rake db:create
```

Código 2.2.5.2 Comando para generar base de datos

2.2.6 Modelo

Para generar un modelo dentro del proyecto se tiene un comando específico, mediante el cual podemos indicar los campos del modelo y sus tipos. Cabe resaltar que el nombre del modelo es recomendable que esté escrito en su forma singular ya que el generador usará esta forma.

```
$ rails generate model NOMBRE_DEL_MODELO CAMPO:TIPO CAMPO:TIPO
```

Código 2.2.6.1 Comando general para crear modelos

Ejemplo para generar un modelo de cursos que tendrá tres campos, *name* de tipo cadena, *start* *y end* de tipo fecha.

```
$ rails generate model Course name:string start:datetime end:datetime
```

Código 2.2.6.2 Ejemplo de uso de comando para generar modelos

Una vez terminada la ejecución del comando genera archivos en los directorios *db/migrate/* y *app/models/* (ver Figura 2.2.6.1) mediante los cuales podemos mapear los campos de los modelos a la base de datos, agregar validaciones, agregar relaciones entre modelos, etc.

```
invoke  active_record
       create    db/migrate/20171016035410_create_courses.rb
       create    app/models/course.rb
```

Figura 2.2.6.1 Información de salida del comando para generar modelos

2.2.6.1 Migraciones

Las migraciones son los archivos mediante los cuales se modifica el esquema de la base de datos, cuando se genera un modelo se agrega una migración (ver Código 2.2.6.1.1) que genera una tabla en la base de datos en la cual definimos los campos junto con sus tipos de datos.

```ruby
class CreateCourses < ActiveRecord::Migration
  def change
    # Se crea una tabla para manejar los cursos
    create_table :courses do |t|
      # Se agrega un campo llamado name de tipo cadena
      t.string   :name
      # Se agrega un campo llamado start de tipo fecha
      t.datetime :start
      # Se agrega un campo llamado end de tipo fecha
      t.datetime :end
      # Se agregan los campos updated_at y created_at
      # que son manejados automáticamente por Ruby on Rails
      t.timestamps null: false
    end
  end
end
```

Código 2.2.6.1.1 Archivo de configuración para migración de cursos

Para generar una migración que modifique una ya existente se tiene una convención en cuanto al nombre la cual es "Add[Campo]To[Tabla]" (ver Código 2.2.6.1.2) en caso de querer agregar un campo y "Remove[Campo]From[Tabla]" para quitar campos de la tabla.

```
$ rails generate migration AddDescriptionToCourses description:text
```

Código 2.2.6.1.2 Agregar migración

Para hacer efectivos los cambios en la base de datos se necesita ejecutar el comando del Código 2.2.6.1.3.

```
$ rake db:migrate
```

Código 2.2.6.1.3 Aplicar cambios de migraciones

Este comando también modifica el archivo *schema.rb* el cual se recomienda usar para inicializar la base de datos en otros equipos, para cargar el esquema se utiliza el comando del Código 2.2.6.1.4.

```
$ rails db:setup
```

Código 2.2.6.1.4 Cargar esquema de base de datos

2.2.6.2 Validaciones

Ruby on Rails permite hacer validaciones desde los modelos lo cual asegura que sólo la información que cumpla ciertos requisitos se va a guardar, si bien hay personas que prefieren evitar hacer estas validaciones ya que exponen la base de datos. Por ejemplo, si se quiere que un campo esté presente al momento de almacenar un objeto a la base de datos (ver Código 2.2.6.2.1).

```
class Course < ApplicationRecord
  validates :name, presence: true, allow_blank: false
end
```

Código 2.2.6.2.1 Validación de presencia de un campo antes de almacenar

2.2.6.3 Operaciones

Ruby on Rails hace uso de *Active Record*[5] para manejar los objetos de la base de datos, es de bastante ayuda ya que permite guardar y obtener objetos de la base de datos sin implementar sentencias *SQL*, así como poder manipular los objetos de la base de datos como objetos de *Ruby*.[3]

2.2.6.3.1 Crear

Para crear un un objeto hay dos maneras, una en la cual se crea la instancia del objeto y se guarda automáticamente y otra en la cual se crea la instancia y se espera hasta que el desarrollador haga la llamada a la función para guardar el objeto en la base de datos.

La ventaja de usar la función *create* es que sólo ocupa una línea (ver línea 2 de Código 2.2.6.3.1.1) aunque tiene la desventaja de que si ocurre un error igualmente regresa una instancia del objeto. La ventaja de usar la función *save* es que regresa un valor de tipo booleano que indica si se pudo guardar el objeto o no, esto sirve cuando una parte del código depende de que se guarde correctamente un objeto (ver líneas 4, 5 y 6 de Código 2.2.6.3.1.1).

[5] Para mayor información https://guides.rubyonrails.org/active_record_basics.html

```
1. # Crea objeto y guarda
2. course = Course.create(name: "Ingeniería de Software II")
3. # Crea objeto y espera hasta que desarrollador guarde
4. course = Course.new
5. course.name = "Ingeniería de Software II"
6. course.save
```

Código 2.2.6.3.1.1 Crear y almacenar objetos

2.2.6.3.2 Leer

Para obtener los objetos que están guardados en la base de datos se tienen varias formas, en las cuales se pueden obtener una colección de objetos o sólo un objeto.

```
# Obtiene todos los objetos de la base
courses = Course.all
# Obtiene el primer objeto de la base
course = Course.first
# Obtiene el objeto cuya llave es igual a la pasada como parámetro
course = Course.find(1)
# Obtiene el primer objeto que cumple con el criterio de búsqueda
course = Course.find_by(name: "Ingeniería de Software II")
# Obtiene todos los objetos que cumplen con los criterios de búsqueda
courses = Course.where(name: "Ingeniería de Software II")
```

2.2.6.3.3 Actualizar

Los objetos de la base de datos se pueden actualizar uno por uno (ver líneas 1 a 7 de Código 2.2.6.3.3.1) o varios elementos al mismo tiempo (ver líneas 8 a 11 de Código 2.2.6.3.3.1), la ventaja de usar la función de *Active Record* para actualizar varios elementos al mismo tiempo con el objetivo de mejorar el rendimiento de las actualizaciones[6].

[6] https://apidock.com/rails/ActiveRecord/Relation/update_all

```
1. # Obtiene y modifica el objeto, espera hasta que desarrollador guarde
2. course = Course.find_by(name: "Ingeniería de Software I")
3. course.name = "Ingeniería de Software II"
4. course.save
5. # Modifica el objeto en el instante
6. course = Course.find_by(name: "Ingeniería de Software I")
7. course.update(name: "Ingeniería de Software II")
8. # Modifica todos los objetos
9. Course.update_all(name: "Ingeniería de Software")
10. # Modifica los objetos cuyo nombre contenga la palabra "Soft"
11. Course.where('name LIKE ?', '%Soft%').update_all(name: "Ingeniería de Software II")
```

Código 2.2.6.3.3.1 Actualizar registros de la base de datos

2.2.6.3.4 Eliminar

Existen dos formas de eliminar los objetos de la base de datos, una que es eficiente (ver líneas 1 a 5 de Código 2.2.6.3.4.1) ya que ejecuta una sentencia *SQL* pero no ejecuta las funciones almacenadas[7] ya que no crea instancias de los objetos *(como borrar dependencias entre otras)* y otra que es menos eficiente (ver líneas 6 a 10 de Código 2.2.6.3.4.1) pero sí ejecuta las funciones[8] ya que sí genera las instancias de los objetos.

```
1. # Obtiene y elimina el objeto de forma eficiente
2. course = Course.find_by(name: "Ingeniería de Software II")
3. course.delete
4. # Elimina todo los objetos cuyo nombre contenga la palabra "Software"
5. Course.delete_all("name LIKE ?', '%Software%")
6. # Obtiene y elimina el objeto
7. course = Course.find_by(name: "Ingeniería de Software II")
8. course.destroy
9. # Elimina todo los objetos cuyo nombre contenga la palabra "Software"
10. Course.destroy_all("name LIKE ?', '%Software%")
```

Código 2.2.6.3.4.1 Eliminar registros de la base de datos

2.2.6.4 Relaciones de modelos

Las relaciones entre modelos simplifican el código ya que permiten realizar operaciones *CRUD* (*Create, Read, Update, Delete; que en español se traduce como Crear, Leer,*

[7] https://apidock.com/rails/ActiveRecord/Relation/delete
[8] https://apidock.com/rails/ActiveRecord/Relation/destroy

Actualizar y Borrar) desde un objeto de un modelo hacia otro objeto de un modelo diferente sin tener que hacer consultas por cada modelo.

Por ejemplo, suponga que se tienen dos modelos uno que representa cursos y otro a profesores y se quieren obtener todos los cursos que son impartidos por el profesor de nombre *Javier Sánchez*. Si no se manejan relaciones entre modelos se tienen que hacer consultas por modelo, es decir, primero obtener el objeto que representa al profesor *Javier Sánchez* y luego buscar los cursos que fueron creados por él.

```
teacher = Teacher.find_by(name: 'Javier Sánchez')
courses = Course.where(teacher_id: teacher)
```

Código 2.2.6.4.1 Obtener cursos de profesor sin manejo de relaciones entre modelos

Usando la relación que se ve en las líneas 1 a 6 del Código 2.2.6.4.2 en donde un profesor puede tener muchos cursos y un curso pertenece a un profesor, la búsqueda se reduce a una línea, línea 7 de Código 2.2.6.4.2.

```
1. class Teacher < ActiveRecord::Base
2.   has_many :courses, dependent: :destroy
3. end
4. class Course < ActiveRecord::Base
5.   belongs_to :teacher
6. end
7. courses = Teacher.find_by(name: X).courses
```

Código 2.2.6.4.2 Manejo de relaciones entre modelos y obtención de cursos

2.2.7 Controlador

Para agregar un controlador dentro del proyecto se tiene un comando mediante el cual podemos indicar las funciones que se van a utilizar para que de este modo agregue las rutas de las funciones al archivo *config/routes.rb*, así como los archivos de las vistas.

```
$ rails generate controller Courses index show --no-test-framework
```

Código 2.2.7.1 Iniciar controlador de cursos

Terminada la ejecución se crea el archivo del controlador con las firmas de las funciones indicadas (ver Figura 2.2.7.1), de igual manera se agregan estas firmas al archivo de rutas

mediante el método *GET*[9] del protocolo *HTTP*[10]. También se genera un archivo con extensión *ERB* para diseñar las vistas de las funciones, estos son plantillas de *HTML* a las que se les agrega código en *Ruby* que el servidor se encarga de transformar en algo que pueda ser entendido por el navegador.

```
     create  app/controllers/course_controller.rb
      route    get 'course/show'
      route    get 'course/index'
     invoke  erb
     create    app/views/course
     create    app/views/course/index.html.erb
     create    app/views/course/show.html.erb
     invoke  helper
     create    app/helpers/course_helper.rb
     invoke  assets
     invoke    coffee
     create      app/assets/javascripts/course.coffee
     invoke    scss
     create      app/assets/stylesheets/course.scss
```

Figura 2.2.7.1 Salida de comando para generar controladores

2.2.7.1 Funciones de controladores

Por lo regular un controlador tiene las siguientes siete funciones (ver Código 2.2.7.1.1): *index, create, new, edit, show, update, destroy;* aunque no se limitan a estas.

[9] Método de HTTP que sirve para obtener información
[10] Protocolo de transferencia de hipertexto

```ruby
# Esta función devuelve todos los registros de la tabla de cursos
def index
end
# Esta función inicializa un objeto de tipo curso
def new
end
# Obtiene la información del curso mediante el parámetro designado (:id)
# y la agrega a la vista
def edit
end
# Guarda el curso en la base de datos
def create
end
# Obtiene el curso mediante el parámetro designado (:id) y lo actualiza
def update
end
# Obtiene el curso mediante el parámetro designado (:id) y lo elimina
def destroy
end
```

Código 2.2.7.1.1 Firmas de las funciones dentro de controlador

2.2.7.2 Rutas

Para manejar las rutas del proyecto se debe modificar el archivo *routes.rb* dentro de la carpeta *config/,* el orden en el que se agreguen las rutas es importante ya que tienen mayor prioridad las rutas que se definen primero (ver Código 2.2.7.2.1).

```
Rails.application.routes.draw do
  # Con la instrucción "root" se indica la página que se regresa cuando
  # se accede a la raíz del sitio e.j misitio.com/
  root 'welcome#index'
  # Para definir una ruta primero se pone el método HTTP y luego la dirección,
  # de esta forma automáticamente se asocia la ruta a la función "index" del
  # controlador "courses"
  get 'courses/index'
  # También se puede indicar de forma manual la función y el controlador,
  # el parámetro as: indica el nombre de una variable mediante la cual se puede
  # hacer uso de la ruta
  get 'courses' => 'courses#index', as: :courses
  # Con la instrucción resources se agregan todos los métodos del protocolo HTTP,
  # con las firmas de funciones convencionales
  resources :courses
end
```

Código 2.2.7.2.1 Archivo de configuración de rutas

2.2.8 Scaffold

Existe otro comando (ver Código 2.2.8.1) que nos permite generar modelos y controladores al mismo tiempo, pero a diferencia de la generación de un controlador en donde sólo se agregan las firmas de las funciones con este comando también se agrega la funcionalidad básica de esas funciones. Las funciones que se generan son las conocidas como operaciones *CRUD*.

```
$ rails generate scaffold NOMBRE_RECURSO CAMPO:TIPO CAMPO:TIPO
```

Código 2.2.8.1 Comando scaffold

En este capítulo se cubrieron nociones básicas de *Ruby y Ruby on Rails*, desde cómo hacer la instalación de cada una a cómo utilizar comandos tanto para iniciar el proyecto como para agregar archivos de configuración y hacer el manejo de las información. Esto con el fin de que se tenga una noción básica del funcionamiento de *Ruby on Rails* para las partes siguientes.

Capítulo 3. Ambiente de desarrollo

3.1 Descripción

Preparar el ambiente de desarrollo para un equipo lleva más tiempo del que se esperaría ya que los equipos/computadoras de cada integrante tienen diferentes características, siendo el sistema operativo uno de los más importantes. Si bien *Ruby on Rails* tiene el archivo *Gemfile* en donde se especifican las versiones de las gemas a usar, no se tiene la certeza que se esté utilizando la misma versión de software independiente como el Sistema Manejador de Bases de Datos (SMBD). Ésta es una de las razones por las que dentro de la comunidad de desarrollo se recomienda[11] utilizar herramientas como *Docker*, que permite a los integrantes del equipo tener la misma configuración del proyecto.

Otro problema es la actualización de un proyecto donde cada integrante pueda ver los cambios que otros integrantes han hecho. Una forma de hacerlo es mediante un sistema de carpetas que distingan las versiones del proyecto y compartir las carpetas con los demás integrantes. Lo anterior conlleva un mayor esfuerzo y tiempo, ya que cada integrante tiene que buscar y actualizar los archivos modificados. Una forma de agilizar este proceso es mediante un controlador de versiones, como *Git*.

Como se puede ver en la Tabla 3.1.1 para nuestro ambiente de desarrollo se utiliza *Git* como controlador de versiones, *PostgreSQL* como SMDB, *Heroku* como servidor de pruebas, etcétera.

Funcionalidad	Nombre	Versión
Controlador de Versiones	*Git*	2.5.2
Sistema Manejador de Bases de Datos	*PostgreSQL*	9.3.9
Lenguaje de Programación	*Ruby*	2.1.5
Marco de Trabajo para Web	*Ruby on Rails*	4.2.3

[11] https://www.docker.com/why-docker, https://www.linode.com/docs/applications/containers/when-and-why-to-use-docker/

Contenedor	*Docker*	17.09
Alojamiento de Repositorio	*GitLab*	--
Alojamiento de Pruebas	*Heroku*	--

Tabla 3.1.1 Ambiente de desarrollo

3.1.1 Controlador de versiones y Alojamiento de Repositorio

Se recomienda tener el proyecto en un repositorio remoto alojado en *GitLab* ya que permite crear repositorios privados que sólo otorgan el acceso a los miembros del equipo. Si no importa quien tenga acceso el proyecto se recomienda utilizar *GitHub* ya que tiene una interfaz más simple y tiene una gran cantidad de herramientas que ayudan en el desarrollo de software como Travis CI para integración continua, Sentry que hace reportes cuando la sistema falla o ImgBot que optimiza el tamaño de las imágenes que tenga tu proyecto.

Cada miembro del equipo tendrá una copia del repositorio principal donde realizará sus cambios, una vez que termine su parte se hará una solicitud para agregar dichos cambios al repositorio principal. Trabajando de esta forma se busca mejorar la calidad del código puesto que el código tiene que ser revisado por todos los integrantes del equipo, así cada uno da su opinión del código hasta que ya no haya observaciones y los cambios sean agregados a la rama *master* del repositorio general.

3.1.1.1 Comandos básicos de *Git*

3.1.1.1.1 Configurar *Git*

Si es la primera vez que se usa *Git* se tienen que configurar el nombre de usuario y el correo, esto es debido a que se hace uso de esta información cuando se hace un *commit.* El nombre de usuario va dentro de comillas dobles para evitar problemas con los espacios entre palabras. [4]

```
$ git config --global user.name "Nombre de usuario"
```

Código 3.1.1.1.1.1 Configuración de nombre de usuario

```
$ git config --global user.email correo@prueba.com
```

Código 3.1.1.1.1.2 Configuración de correo electrónico

3.1.1.1.2 Iniciar repositorio local

Para iniciar un repositorio local está el comando *init* (Código 3.1.1.1.2.1) que crea la carpeta *.git/* que es donde se almacena todo lo referente al repositorio local, como el registro de *commits*, la dirección del repositorio remoto, comandos que se deben ejecutar antes, durante o después tanto de confirmar cambios como de subirlos al repositorio remoto. [4]

```
$ git init
```

Código 3.1.1.1.2.1 Iniciar repositorio local

3.1.1.1.3 Estado del repositorio

Existe un comando que permite conocer los archivos que han sido creados, modificados, renombrados o removidos desde el último *commit.* [4]

```
$ git status
```

Código 3.1.1.1.3.1 Obtener estado del repositorio

3.1.1.1.4 Manejo de ramas

Lo recomendable es crear ramas en donde se hace el desarrollo del proyecto y en la rama maestra *(master)* tener el código que ha sido revisado y aprobado por los miembros del equipo. [4]

3.1.1.1.4.1 Crear rama

Cuando se crea una rama se recomienda asignarle un nombre que identifique la funcionalidad a desarrollar, en el Código 3.1.1.1.4.1 se muestra cómo se puede crear una rama, mientras que en el Código 3.1.1.1.4.2 se muestra cómo se puede crear una rama y moverse a ella en el mismo comando, en este caso el comando *checkout* es el que se encarga de cambiar de rama y la bandera *-b* es la que dice que se cree una nueva rama.

```
$ git branch inicio_de_sesion
```

Código 3.1.1.1.4.1 Creación de rama

```
$ git checkout -b inicio_de_sesion
```

Código 3.1.1.1.4.2 Creación y cambio de rama

3.1.1.1.5 Guardar cambios

Los archivos de un repositorio local se pueden modificar sin embargo no se verán reflejados en el repositorio remoto hasta que se confirmen los cambios y se suban al repositorio remoto. Para guardar los cambios de un proyecto primero se tienen que escoger los archivos que se quieren actualizar y después confirmar los cambios añadiendo un texto que indique los cambios que se hicieron.

Para indicar los archivos que se quieren actualizar está el comando *add*, en el cual se puede indicar uno por uno los archivos (ver Código 3.1.1.1.5.1) o usar el símbolo "." (ver Código 3.1.1.1.5.2) para indicar todos los archivos de la carpeta sobre la que se está trabajando.[4]

```
$ git add archivo_1 archivo_2 archivo_3
```

Código 3.1.1.1.5.1 Actualización *Git* archivo por archivo

```
$ git add .
```

Código 3.1.1.1.5.2 Actualización *Git* todos los archivos de la carpeta

Para confirmar los cambios se tiene la forma compacta en donde sólo se pone un mensaje corto (ver Código 3.1.1.1.5.3) que sirve para conocer los qué cambios se realizaron o la forma extendida (ver Código 3.1.1.1.5.4), que abre el editor de texto predeterminado en donde aparte del mensaje corto se puede agregar un texto con una descripción más larga que especifique a detalle lo que se trabajó. [4]

```
$ git commit -m "Se agrega ..."
```

Código 3.1.1.1.5.3 Confirmar cambios (compacta)

```
$ git commit
```

Código 3.1.1.1.5.4 Confirmar cambios (extendida)

3.1.1.1.6 Subir cambios a repositorio remoto

Para subir los cambios hechos en el repositorio local a un repositorio remoto se encuentra el comando *push*, sus parámetros son el nombre del repositorio al que se quieren subir los cambios y la rama que se quiere actualizar. La bandera *-u* le dice a *Git* que recuerde los parámetros (ver Código 3.1.1.1.6.1) para que a la próxima vez que se haga un *push* ya no sea necesario escribirlos (ver Código 3.1.1.1.6.2).[4]

```
$ git push -u origin master
```

Código 3.1.1.1.6.1 Subir cambios a repositorio remoto con parámetros

```
$ git push
```

Código 3.1.1.1.6.2 Subir cambios a repositorio remoto sin parámetros

3.1.1.1.7 Bajar cambios de repositorio remoto

Para actualizar una rama del repositorio local con cambios del repositorio remoto se tiene el comando *pull*, sus parámetros son el nombre del repositorio del que se quieren obtener los cambios y la rama que se quiere descargar (ver Código 3.1.1.1.7.1). Al usar este comando se pueden generar conflictos entre archivos modificados localmente y remotos, para solucionar los conflictos se tiene que modificar archivo por archivo (sólo los que tuvieron conflictos) y guardar los cambios.[4]

```
$ git pull origin master
```

Código 3.1.1.1.7.1 Bajar cambios de rama *master* del repositorio remoto *origin*

3.1.1.1.8 Esconder cambios

Hay veces que se están haciendo cambios en el repositorio local pero se necesita solucionar un error con urgencia en el repositorio remoto, para no perder estos cambios se podría hacer un *commit* pero puede que sean cambios minúsculos que no lo justifiquen. Para eso está el comando *stash* (ver Código 3.1.1.1.8), para esconder los

cambios que se tienen, hacer lo que se tenga que hacer y después recuperar los cambios para continuar con lo que se estaba haciendo.[4]

```
$ git stash
```

Código 3.1.1.1.8.1 Esconder cambios repositorio local

3.1.1.1.9 Enlistar cambios escondidos

Este comando (ver Código 3.1.1.1.9.1) muestra si hay cambios escondidos en el repositorio local.[4]

```
$ git stash list
```

Código 3.1.1.1.9.1 Enlistar cambios escondidos

3.1.1.1.10 Mirar cambios escondidos

Este comando (ver Código 3.1.1.1.10.1) obtiene los cambios escondidos y muestra los archivos que han sido modificados, se puede pasar como parámetro el índice (ver Código 3.1.1.1.10.2) al cambio especificado en caso contrario mostrará la información del último cambio escondido.[4]

```
$ git stash show
```

Código 3.1.1.1.10.1 Mostrar archivos modificado en cambios escondidos

```
$ git stash show stash@{2}
```

Código 3.1.1.1.10.1 Mostrar archivos modificado en cambios escondidos índice 2

3.1.1.1.11 Recuperar cambios escondidos

Este comando remueve y aplica un cambio escondido, recibe de manera opcional el parámetro *<stash>*, de la forma *stash@{<número>}* (ver Código 3.1.1.1.11.2) o pasando sólo el número, en caso de ser proporcionado se hará la recuperación de los cambios especificados en caso contrario tomará el último cambio escondido (ver Código 3.1.1.1.11.1).[4]

```
$ git stash pop
```

Código 3.1.1.1.11.1 Recuperar último cambio escondido

```
$ git stash pop stash@{1}
```

Código 3.1.1.1.11.2 Recuperar cambio escondido de índice 1

3.1.2 *Docker*

Como se expuso al inicio de este capítulo muchas veces el desarrollo de software en equipo conlleva una gran cantidad de tiempo en la configuración del ambiente de desarrollo, ya sea porque no todas las personas tienen la misma versión del SMBD o de algún lenguaje de programación como *Python, Java o Ruby* ya que pueden estar en más de un proyecto y esos proyectos requieren otra versión o por alguna otra razón. Para tratar de mitigar estos tiempos se recomienda el uso de *Docker*, que permite generar imágenes que pueden ser por compartidas entre los miembros del equipo sin importar las versiones de software que tengan.

Una imagen es un conjunto de capas que definen el comportamiento de un contenedor, un contenedor es una imagen que ha sido o está siendo ejecutada.[12]

Para poder usar *Docker* se necesita tener el demonio *dockerd* activo para que el cliente de *Docker* pueda trabajar, en este documento se mostrará cómo activar el demonio[13] con *systemctl*[14] (ver Código 3.1.2.1) y una forma alternativa mediante el mismo *dockerd* que permite seguir los registros en la consola (ver Código 3.1.2.2).

```
$ systemctl start dockerd.service
```

Código 3.1.2.1 Demonio *dockerd*

```
$ dockerd
```

Código 3.1.2.2 Iniciar dokcerd desde línea de comandos

En algunos casos se requiere ejecutar los comandos con privilegios, en esos casos agregar el comando *sudo* (ver Código 3.1.2.3) antes de cada uno.

[12] https://docs.docker.com/v17.09/get-started/#a-brief-explanation-of-containers

[13] Proceso que se corre en segundo plano que responde a solicitudes de servicios http://www.linfo.org/daemon.html

[14] Utilidad de administración que combina las herramientas *service* y *chkconfig* de SysV, por lo tanto podremos arrancar, parar, recargar servicios, activar o desactivar servicios en el arranque, listar los estados de los servicios, etcétera.

```
$ sudo dockerd
```

Código 3.1.2.2 Iniciar dokcerd desde línea de comandos con privilegios

A continuación se mostrarán una serie de comandos que serán de ayuda durante el desarrollo con *Docker*, para esto se ocupa el cliente de *Docker* el cual se ejecuta como *Docker* en la línea de comandos. Se utiliza la bandera *-h* para obtener más información acerca del comando (ver Código 3.1.2.3).

```
$ docker -h
```

Código 3.1.2.3 Obtener información acerca del comando *docker*

3.1.2.1 Generar imagen

La opción *build* indica que se quiere construir una imagen a partir de un archivo *Dockerfile*, la bandera *-t* sirve para nombrar la imagen (ver Código 3.1.2.1.1) y opcionalmente ponerle una etiqueta (ver Código 3.1.2.1.2) que sirve para diferenciar entre versiones, finalmente se pasa la ruta del directorio en el cual está el archivo *Dockerfile*.

```
$ docker build -t rails_app .
```

Código 3.1.2.1.1 Generar y nombrar imagen de *Docker*

```
$ docker build -t rails_app:latest .
```

Código 3.1.2.1.2 Generar, nombrar y versionar imagen de *Docker*

En caso de éxito se obtendrá un resultado de este estilo ***Successfully tagged rails_app:latest***, se pueden ver más opciones para construir las imágenes pasando la bandera *-h* (ver Código 3.1.2.1.3).

```
$ docker build -h
```

Código 3.1.2.1.3 Mostrar opciones para generar una imagen de *Docker*

3.1.2.2 Enlistar imágenes

Para obtener las imágenes más importantes se tiene el comando del Código 3.1.2.2.1.

Al utilizar la bandera *-a* se indica que se deben de enlistar todas las imágenes, esto incluye las imágenes intermedias que se descargan o generan de otras imágenes (ver Código 3.1.2.2.2).

```
$ docker images
```

Código 3.1.2.2.1 Mostrar imágenes

```
$ docker images -a
```

Código 3.1.2.2.2 Mostrar todas las imágenes de *Docker*

3.1.2.3 Remover imagen

Para remover una imagen primero hay que estar seguros que no hay un contenedor activo que se base en la imagen, en caso de haber uno primero hay que detener el contenedor y después borrar la imagen en base a su identificador (ver Código 3.1.2.3.1). Si lo que se necesita es remover todas las imágenes hay una serie de comandos que facilitan esta tarea (ver Código 3.1.2.3.2).

```
$ docker rmi IMAGE_ID
```

Código 3.1.2.3.1 Remover imagen en base a su identificador

```
$ docker rmi $(docker images -aq)
```

Código 3.1.2.3.2 Remover todas las imágenes inactivas

3.1.2.4 Crear un contenedor

Para crear y tener activo un contenedor se ejecuta el siguiente comando, donde *-p 8000:3000* indica que el puerto 3000 del contenedor se va a asociar con el puerto 8000 de la máquina anfitriona, la bandera *-d* indica que el contenedor se va a seguir ejecutando en segundo plano y por el último está el nombre de la imagen que va a usar el contenedor (ver Código 3.1.2.4.1).

```
$ docker run -p 8000:3000 -d dockerrails_rails
```

Código 3.1.2.4.1 Crear contenedor y asociar puertos, ejecución en segundo plano

3.1.2.5 Enlistar contenedores

Existe un comando que permite obtener todos los contenedores activos, esto sirve para verificar que un contenedor que se inició en segundo plano esté activo (ver Código 3.1.2.5.1).

```
$ docker ps
```

Código 3.1.2.5.1 Obtener todos los contenedores activos

Al igual que con las imágenes se puede pasar la bandera *-a* la cual en este caso nos mostrará tanto los contenedores que están activos como los que ya no lo están.

3.1.2.6 Detener un contenedor

Cuando se crea un contenedor en primer plano se puede detener simplemente terminado la ejecución del proceso, pero cuando el contenedor está en segundo plano hay que usar el comando del Código 3.1.2.6.1 para pararlo.

```
$ docker stop CONTAINER_ID CONTAINER_ID
```

Código 3.1.2.6.1 Detener un contenedor

Si se requiere detener todos los contenedores se debe de escribir el siguiente comando el cual se encarga de buscar todos los contenedores y detener su ejecución (ver Código 3.1.2.6.2).

```
$ docker stop $(docker ps -aq)
```

Código 3.1.2.6.2 Detener ejecución de todos los contenedores

En donde la bandera *-q* indica que sólo se obtengan los identificadores de los contenedores y la bandera *-a* que muestre todos los contenedores, incluso los que no estén activos.

3.1.2.7 Remover un contenedor

Se recomienda eliminar los contenedores que no se utilizan ya que estos ocupan espacio y recursos del sistema. Al igual que el comando para detener los contenedores se puede pasar la lista de identificadores de los contenedores a remover (ver Código 3.1.2.7.1) o se puede para que remueva todos de una sola vez (ver Código 3.1.2.7.2).

```
$ docker rm e4e4b655d859
```

Código 3.1.2.7.1 Remover contenedores específicos

```
$ docker rm $(docker ps -aq)
```

Código 3.1.2.7.2 Remover todos los contenedores

3.1.2.8 Comandos dentro del contenedor

Un tema importante es cómo ejecutar un comando dentro de un contenedor, por ejemplo en Ruby on Rails es necesario ejecutar las migraciones cuando se hacen cambios en los modelos, para esto se tiene el subcomando *exec* que nos permite ejecutar comandos en contenedores activos (ver Código 3.1.2.8.1).

Las banderas *-i y -t* son obligatorias en caso de querer el *intérprete de comandos (shell)* para ejecutar diferentes comandos (ver Código 3.1.2.8.2), en caso de querer sólo un comando que no implique interacción se pueden omitir (ver Código 3.1.2.8.3). Siendo más específico, la bandera *-i* mantiene el flujo de datos de entrada abierto y la bandera *-t* permite el uso de una pseudo terminal.

```
$ docker exec -it CONTAINER_ID COMANDO
```

Código 3.1.2.8.1 Comando dentro de contenedor con flujo de datos abierto

```
$ docker exec -it a768943facf2 bash
```

Código 3.1.2.8.2 Ejemplo de comando dentro de contenedor con flujo de datos abierto

En este caso no es necesario mantener el flujo de datos abierto ya que sólo se requiere obtener la lista de archivos dentro del directorio actual (ver Código 3.1.2.8.3).

```
$ docker exec a768943facf2 ls
```

Código 3.1.2.8.3 Ejecutar comando dentro de un contenedor sin flujo de datos

3.1.2.9 Volúmenes de información

La información que se tiene dentro de un contenedor desaparece en el momento en el que éste se detiene, por lo que si se tiene un contenedor con la base de datos y se llegará a detener se perderían todos los registros que se tenían hasta el momento. Para casos como éste se hace el uso de volúmenes (ver Código 3.1.2.9.1 y Código 3.1.2.9.2) y asociación de montajes que si bien son relativamente parecidos se recomienda utilizar los volúmenes ya estos no dependen de la estructura de carpetas de la máquina anfitriona mientras que la asociación de montajes sí, aunque ésta es de gran ayuda para actualizar el código de un proyecto dentro de un contenedor ya que se actualiza al mismo tiempo.

```
$ docker run -v NOMBRE_VOLUMEN:DIRECCION_DENTRO_DE_CONTENEDOR IMAGEN
```

Código 3.1.2.9.1 Forma general de creación de volumen en *Docker*

```
$ docker run -v postgres_data:/var/lib/postgresql/data postgres
```

Código 3.1.2.9.2 Ejemplo de creación de volumen en Docker

3.1.3 *Dockerfile*

El archivo que maneja *Docker* se llama *Dockerfile*, éste es un script que contiene varias instrucciones y una imagen base que ya existe con el fin de crear una nueva imagen que pueda ser distribuida entre un equipo de desarrollo.

A continuación se explica un poco acerca de las instrucciones más utilizadas dentro de un *Dockerfile*[15].

- Todo *Dockerfile* inicia con la instrucción *FROM,* la cual indica cuál va a ser la imagen base de la nueva imagen.
- La instrucción LABEL no es más que una instrucción informativa, con ésta podemos definir la versión, autores, descripción, etc.
- La instrucción EXPOSE sirve para informar que la imagen generada se va a asociar con el puerto especificado por lo que al momento de activar el contenedor se debe de asociar a ese puerto.
- La instrucción ENV sirve para definir variables de ambiente.

[15] En el siguiente enlace se muestran más instrucciones disponibles para definir un *Dockerfile* https://docs.docker.com/v17.09/engine/userguide/eng-image/dockerfile_best-practices/#the-dockerfile-instructions

- La instrucción WORKDIR sirve para especificar el directorio dentro del cual se va a trabajar, esta instrucción puede utilizarse cuantas veces sea necesaria.
- Con las instrucciones ADD/COPY indicamos que queremos copiar un archivo o directorio de la máquina anfitriona al contenedor, aunque los dos comandos trabajan de forma similar se recomienda el uso de COPY ya que éste sólo hace la copia de los archivos, mientras que ADD puede realizar cosas que no se le indican cómo descomprimir automáticamente un archivo .tar dentro del contenedor.
- La instrucción RUN sirve para ejecutar comandos dentro del contenedor como instalar paquetes.
- La instrucción CMD es la que se encarga de correr lo que esté en la imagen, como un servidor.

A continuación un ejemplo de un archivo *Dockerfile* (ver Código 3.1.3.1), el cual está basado en una imagen de *Ruby* en su versión 2.1.5, que copia un proyecto de *Ruby on Rails*, expone el puerto 3000 e inicia el servidor sobre la dirección 0.0.0.0.

```
FROM ruby:2.1.5

LABEL authors="Jesús <jvila@ciencias.unam.mx>"
LABEL version="1.0"
LABEL description="Imagen básica para proyecto de RoR"

EXPOSE 3000
ENV app /usr/src/app

WORKDIR ${app}
COPY app .
RUN bundle install

CMD ["rails", "server", "-b", "0.0.0.0"]
```

Código 3.1.3.1 Archivo de configuración *Dockerfile*

3.1.4 *Docker Compose*

Docker Compose[16] es una herramienta que nos permite definir múltiples imágenes y crear contenedores de éstas con el comando *docker-compose*.

[16] En algunos sistemas operativos basta instalar *Docker* para tener *Compose,* en otros es necesario instalar por separado: https://docs.docker.com/compose/install/#install-compose

Para utilizar la herramienta *Compose* se necesita definir un archivo con formato *YAML (YAML Ain't Markup Language)* para la serialización de datos llamado *docker-compose.yml* o *docker-compose.yaml*, ésta herramienta automáticamente obtiene la información de un archivo que tenga ese nombre, en caso de querer asignar un nombre diferente se necesita pasar la bandera *-f* con el nombre del archivo.

La configuración más sencilla de un archivo *docker-compose.yml* (ver Código 3.1.4.1) consta de la versión y los servicios, a continuación se muestra la configuración para crear un contenedor con la imagen más reciente de *PostgreSQL* que asocia el puerto 5432 del contenedor con el 5432 de la máquina anfitriona.

```
version: '3'

services:
  db:
    image: postgres
    ports:
      - 5432:5432
```

Código 3.1.4.1 Archivo *docker-compose.yml* que inicia un contenedor de *PostgreSQL*

Aunque es común que junto a los servicios también se definan los volúmenes de información y las redes que usan los contenedores. A continuación se muestra el mismo ejemplo pero haciendo uso de redes y volúmenes (ver Código 3.1.4.2). Ahora la información del directorio */var/lib/postgresql/data* se va a almacenar en el volúmen *db_data* para que sea persistente y se va a agregar a la red *backend* para que la base no esté expuesta al público.

```yaml
version: '3'
services:
  db:
    image: postgres
    ports:
      - 5432:5432
   volumes:
     - db_data:/var/lib/postgresql/data
   networks:
     - backend
volumes:
 db_data:
networks:
 backend:
```

Código 3.1.4.2 Archivo *docker-compose.yml* que inicia un contenedor de *PostgreSQL* con volúmenes de información y redes

Los ejemplos anteriores sólo manejan un servicio, sin embargo al usar la herramienta *Compose* se pueden declarar varios servicios para generar todos los contenedores necesarios con un sólo comando. Al igual que con el contenedor para la base de datos sólo tenemos que definir los contenedores dentro del objeto *services.* Por ejemplo, para crear un contenedor con una aplicación que se encuentra en la carpeta *app* y se conecte a la base de datos se puede utilizar la siguiente plantilla. Al igual que en el Código 3.1.4.2 la información del directorio */var/lib/postgresql/data* se va a almacenar en el volúmen *db_data* para que sea persistente y se va a agregar a la red *backend* para que la base no esté expuesta al público. También se agrega el enlace entre el servicio *web* y el servicio *db* que maneja la base de datos.

```
version: '3'
services:
  db:
    image: postgres
   volumes:
     - db_data:/var/lib/postgresql/data
    ports:
      - 5432:5432
  web:
    build: .
    image: rails:latest
    volumes:
      - ./app:/usr/src/app
    ports:
      - 3000:3000
    links:
      - db
volumes:
 db_data:
```

Código 3.1.4.3 Archivo *docker-compose.yml* que inicia un contenedor de la aplicación y se conecta con un contenedor de *PostgreSQL* con volúmenes de información y redes

Al momento de definir un servicio se recomienda poner al principio las configuraciones importantes como los puertos, volúmenes, variables de entorno y redes ya que así se facilita la lectura para las personas que lean la configuración[17].

3.1.4.1 Construir/actualizar imágenes

Docker Compose tiene un comando el cual permite construir las imágenes de los servicios que se definan en el proyecto mediante archivos *Dockerfile*, en este paso se omiten las imágenes de servicios que sólo necesitan descargarse (ver Código 3.1.4.1.1).

---

[17] Para más opciones acerca de la configuración de los servicios consultar el siguiente enlace https://docs.docker.com/compose/compose-file/#service-configuration-reference

Se anexa el enlace a un repositorio de *GitHub* en donde se puede encontrar una plantilla de *Docker* para utilizar https://github.com/vila8/rails-docker.

```
$ docker-compose build
```

Código 3.1.4.1.1 Generar/actualizar imagen de todos los servicios

También se puede especificar cuáles son los servicios que se quieren generar/actualizar (ver Código 3.1.4.1.2)

```
$ docker-compose build wep_app
```

Código 3.1.4.1.2 Generar/actualizar imagen de un servicio

Para generar imágenes de un archivo con nombre diferente de *docker-compose.yml o docker-compose.yaml* se utiliza la bandera *-f* que debe de estar inmediatamente después del comando *docker-compose* y antes de cualquier subcomando (ver Código 3.1.4.1.3).

```
$ docker-compose -f docker-compose.dev.yml build
```

Código 3.1.4.1.3 Generar imágenes a partir de archivo en especifico

3.1.4.2 Crear contenedores

En caso de no ejecutar el comando para construir las imágenes el comando para crear los contenedores (ver Código 3.1.4.2.1) se encarga de ejecutarlo con la diferencia que sí descarga las imágenes de los servicios que usan imágenes de *Docker Hub*[18] https://hub.docker.com/, así como también se encarga de crear y asociar los volúmenes y redes especificadas en el archivo.

```
$ docker-compose up
```

Código 3.1.4.2.1 Crear contenedores con comando *docker-compose*

El comando anterior mantiene en primer plano la ejecución de los contenedores, lo que es de ayuda cuando se está en el proceso de desarrollo pero cuando se está en producción se necesita mandar el proceso a segundo plano (ver Código 3.1.4.2.2) para continuar con otras configuraciones que requiera el proyecto, para eso está la bandera *-d.*

```
$ docker-compose up -d
```

Código 3.1.4.2.2 Crear contenedores y mantener en segundo plano

[18] Repositorio remoto de imágenes de *Docker*

3.1.4.3 Detener contenedores

Para detener los contenedores se pueden utilizar los comandos antes vistos pero el detalle de usarlos es que detienen todos los contenedores activos y no únicamente los que están definidos dentro del *compose*, para detener únicamente estos contenedores *compose* cuenta con el comando *down.* (ver Código 3.1.4.3.1)

```
$ docker-compose down
```

Código 3.1.4.3.1 Detener contenedores asociados al archivo *docker-compose.yml*

Con la bandera *-v* indicamos que a parte de detener los contenedores queremos remover los volúmenes asociados a éstos. (ver Código 3.1.4.3.2)

```
$ docker-compose down -v
```

Código 3.1.4.3.2 Detener contenedores y volúmenes asociados al archivo *docker-compose.yml*

3.1.4.4 Registros del contenedor

Cuando se inician los contenedores en segundo plano no se tiene noción lo que éste imprime en pantalla. Por ejemplo cuando se inicia un servidor de Ruby on Rails se imprimen las peticiones que llegan y las consultas que se hacen a la base de datos, esto es de gran utilidad ya que se verifica que llegue la petición adecuada y se realicen las consultas debidas. Para solucionar este problema se podrían iniciar los contenedores en primer plano y esto nos muestra los registros del contenedor pero como se comentó anteriormente en producción un contenedor se debe de mandar a segundo plano para que se siga ejecutando aun cuando ya no se esté conectado al servidor.

Con el siguiente comando (ver Código 3.1.4.4.1) se obtienen los registros de todos los servicios definidos dentro del archivo *docker-compose.yml,* esto causa que sea difícil leer los registros de un servicio en específico*.*

```
$ docker-compose logs
```

Código 3.1.4.4.1 Comando para obtener todos los registros de un archivo *docker-compose.yml*

Una solución al problema anterior es ejecutar el comando (ver Código 3.1.4.4.2) especificando el nombre del servicio del cual se necesitan los registros.

```
$ docker-compose logs db
```

Código 3.1.4.4.2 Obtener registros del servicio *db*

Ahora el problema es que muestra todos los registros desde que se inició el contenedor y no mantiene el flujo de registros. Para esto se tienen dos bandera en específico, la bandera *-f* indica que continúe el flujo de datos por lo que sólo se ejecuta una vez el comando y la bandera *--tail* que indica el número de registros que se deben mostrar al principio tomando las últimas *n* entradas. En el caso del ejemplo (ver Código 3.1.4.4.3) al principio se muestran las últimas diez entradas de los registros y se mantiene el flujo del servicio *db*.

```
$ docker-compose logs -f --tail=10 db
```

Código 3.1.4.4.3 Obtener las últimas 10 entradas de los registros y mantener flujo

Con las configuraciones y comandos vistos hasta el momento se puede generar una configuración básica, que sirve en caso de querer utilizar *Docker Swarm*[19], una herramienta que permite tener varios equipos ejecutando *Docker* conectados para distribuir tareas y minimizar el uso de procesador así como ayuda a evitar el tiempo de inactividad del servidor al momento de actualizar ya que no renueva todos los equipos conectados mediante *Swarm* al mismo tiempo.

3.1.5 Heroku

Heroku (PaaS) es una plataforma cuyo funcionamiento es parecido al de un servicio web, el sirve para montar aplicaciones web sin costo utilizando la versión más austera, además se puede escalar la aplicación dependiendo de los requerimientos y presupuesto[20]. Lo anterior resulta benéfico pues permite al cliente realizar pruebas sobre el sistema sin tener que instalar todo el software necesario para ejecutar el proyecto[21].

Es necesario crear una cuenta desde la página oficial[22] para poder utilizar la interfaz de línea de comandos (CLI), ya que se ha instalado el CLI es necesario iniciar sesión (ver Código 3.1.5.1) ya que solo así se podrá hacer uso de *Heroku*.

---

[19] Para más información consultar el siguiente enlace así como la documentación que se encuentra en el sitio oficial de *DocKer* https://docs.docker.com/engine/swarm/swarm-tutorial/

[20] https://www.heroku.com/what

[21] Esta plataforma provee una interfaz de línea de comandos (CLI) mediante el cual se hará todo el manejo de la aplicación, actualizar, definir variables de entorno, revisar registros, etcétera.

[22] Para mayor información ir a la siguiente dirección https://signup.heroku.com/login

```
$ heroku login
Enter your Heroku credentials.
Email: <Introducir correo>
Password: <Introducir contraseña>
```

Código 3.1.5.1 Iniciar sesión en interfaz de línea de comandos de *Heroku*

Después de que se haya iniciado sesión ya se pueden crear aplicaciones utilizando el CLI, para iniciar una aplicación se necesita estar en el directorio del proyecto para que automáticamente se agregue el repositorio de *Heroku* a la lista de repositorios remotos del proyecto. Hay dos formas de crear una aplicación en *Heroku* (ver Código 3.1.5.2), la primera es que se genere con un nombre aleatorio como *obscure-fortress-46497* y la segunda es especificando el nombre, aunque hay veces en las que el nombre ya ha sido ocupado por otra aplicación.

```
$ heroku create
$ heroku create nombre_del_proyecto
```

Código 3.1.5.2 Iniciar aplicación en *Heroku*

Se necesita incluir un archivo de nombre *Procfile* en donde se debe de poner explícitamente el comando que se utiliza para ejecutar la aplicación. En el Código 3.1.5.3 se muestra el contenido del archivo si se utiliza la gema *puma* como servidor de la aplicación, cabe resaltar el uso de la palabra *web* que indica que la aplicación recibirá tráfico de la web.

```
web: bundle exec puma -C config/puma.rb
```

Código 3.1.5.3 Configuración de archivo *Procfile*

Cuando ya se ha creado la aplicación se pueden actualizar los código del repositorio remoto utilizando algunos comandos de *Git*, procurando actualizar la rama *master* para que surtan efecto los cambios en el repositorio de *Heroku*.

3.1.5.1 Registros de la aplicación

Utilizando el CLI se puede acceder a los registros de la aplicación (ver Código 3.1.5.1.1) y así poder ver lo que muestra en pantalla la aplicación en caso de que algo esté fallando o se estén haciendo pruebas. Se utiliza la bandera --tail para que sólo muestre las últimas entradas de los registros.

```
$ heroku logs --tail
2017-12-12T23:36:43.249430+00:00 app[web.1]: POST /v1/graphql 200 5.695 ms - 161
2017-12-12T23:36:43.250504+00:00 heroku[router]: at=info method=POST path="/v1/graphql"
host=memorick-node.herokuapp.com         request_id=8c87a9d2-d171-4cff-8940-541db6846a03
fwd="201.149.28.154"    dyno=web.1    connect=0ms    service=7ms    status=200    bytes=364
protocol=https
2017-12-12T23:57:41.139083+00:00 heroku[router]: at=info method=POST path="/v1/graphql"
host=memorick-node.herokuapp.com         request_id=4ee3626a-6641-4281-a27d-bc23e01a49f6
fwd="201.149.28.154"    dyno=web.1    connect=1ms    service=7ms    status=200    bytes=364
protocol=https
```

Código 3.1.5.1.1 Registros de la aplicación en *Heroku*

3.1.5.2 Variables de entorno

El CLI permite agregar variables de entorno a la aplicación, a continuación (ver Código 3.1.5.2.1) se mostrará la forma manual y otra usando la gema *figaro* expuesta anteriormente (ver Código 3.1.5.2.2).

```
$ heroku config:set NOMBRE_VARIABLE=valor
```

Código 3.1.5.2.1 Configuración de variables de entorno manual

```
$ figaro heroku:set -e production
```

Código 3.1.5.2.2 Configuración de variables de entorno mediante figaro

También se pueden ver las variables de entorno definidas (ver Código 3.1.5.2.3), con esto se puede asegurar que los comandos anteriores hayan funcionado como se espera.

```
$ heroku config
```

Código 3.1.5.2.3 Obtener lista de variables de entorno definidas

3.1.5.3 Actualizar

Para actualizar el código que se ejecuta en *Heroku* es necesario hacer un *push* a la rama *master* del repositorio que crea *Heroku*. Aquí se manejan dos casos, hacer *push* de la rama *master* local a la rama *master* del repositorio (ver Código 3.1.5.3.1) o hacer *push* de cualquier otra rama local a la rama *master* del repositorio (ver Código 3.1.5.3.2).

```
$ git push heroku master
```

Código 3.1.5.3.1 Actualización de cambios desde rama *master* local

```
$ git push heroku otra_rama:master
```

Código 3.1.5.3.2 Actualización de cambios desde otra rama local

3.1.5.4 Configuración de Base de Datos en *Heroku*

A continuación se muestra una pantalla donde se puede observar la creación de una base de datos en *Heroku*, primero hay que seleccionar la aplicación y movernos a la pestaña de *recursos (Resources).*

Figura 3.1.5.4.1 Sección de recursos de una aplicación en *Heroku*

En la parte inferior de la pantalla está un campo para buscar complementos, se escribe la palabra *Postgres* y se mostrará el complemento *Heroku Postgres*.

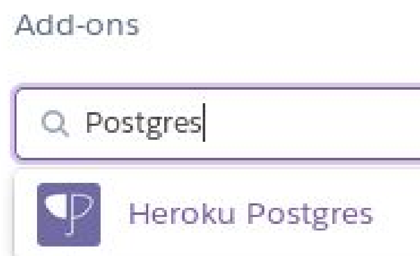

Figura 3.1.5.4.2 Complemento para base de datos *Heroku Postgres*

Al seleccionar éste complemento aparecen las versiones disponibles (ver Figura 3.1.5.4.3) y sólo se escoge la versión requerida. En este caso se utiliza la versión gratuita, *Hobby Dev*, que provee 10,000 registros siendo éstas suficientes para una aplicación pequeña o para un servidor de pruebas[23].

[23] https://devcenter.heroku.com/articles/heroku-postgres-plans#hobby-tier

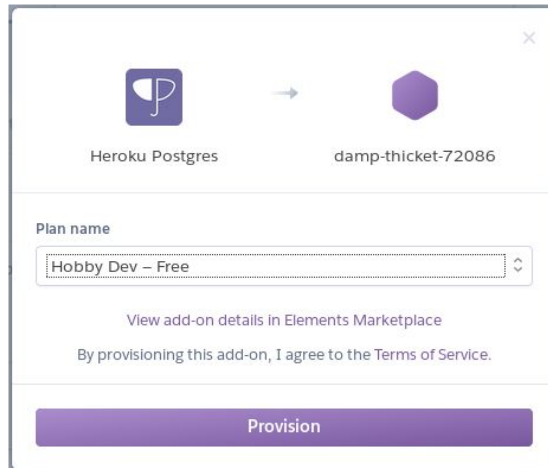

Figura 3.1.5.4.3 Versiones del complemento *Heroku Postgres*

De igual forma se puede utilizar la interfaz de línea de comandos para agregar el complemento (ver Código 3.1.5.4.1).

```
$ heroku addons:create heroku-postgresql:hobby-dev
```

Código 3.1.5.4.1 Complemento para base de datos *Heroku Postgres con CLI*

De forma automática se genera la variable de entorno *DATABASE_URL* parecida a postgres://<usuario>:<contraseña>@<dirección_máquina>:<puerto>/<base_de_datos>

3.2 Desplegar en producción

3.2.1 *Heroku* con *Docker*

Heroku permite desplegar aplicaciones basadas en *Docker*, para ello primero se tiene que iniciar sesión en el *Registro de Contenedores de Heroku* (ver Código 3.2.1.1) (*Heroku Container Registry*) para poder hacer uso de *Docker*.

```
$ heroku container:login
```

Código 3.2.1.1 Iniciar sesión en el *Registro de Contenedores de Heroku*

Al archivo *Dockerfile* se le debe de agregar la instrucción *CMD*[24] *(command)* ya que no se hace uso de la herramienta *Compose* con *Heroku*, este comando no interviene con los servicios de *Compose* en la etapa de desarrollo ya que *Compose* se encarga de

[24] Instrucción que se encarga de correr lo que esté en la imagen, como un servidor.

sobrescribir la instrucción *CMD* con lo que se haya definido en el servicio que utilice ese *Dockerfile*. A continuación se muestra una plantilla (ver Código 3.2.1.2) básica que evita instalar las gemas que se utilizan para desarrollo y pruebas.

```
# Se utiliza la imagen de Ruby en su versión 2.1.5
FROM ruby:2.1.5
# Se agrega la información del autor de la imagen y la versión
LABEL author="Jesús Vila <jvila@ciencias.unam.mx>"
LABEL version="1.5"
# Se actualiza el contenedor y se instalan paquetes para desarrollo
# como nodejs y un cliente de PostgreSQL
RUN apt-get update \
   && apt-get install -qq -y build-essential nodejs \
   libpq-dev postgresql-client --fix-missing --no-install-recommends
# Se agrega una variable de entorno
# que apunta al directorio /usr/src/app
ENV home /usr/src/app
# Se asigna el directorio /usr/src/app como el directorio de trabajo
WORKDIR ${home}
# Se copia el contenido del directorio app del sistema de archivos
local al sistema de archivos de Docker
COPY app .
# Se ejecuta el comando para instalar las gemas del proyecto sin las
# gemas que se utilizan en el entorno de pruebas
RUN bundle install --without development:test
```

Código 3.2.1.2 Archivo *Dockerfile* para *Heroku*

Ya que se tiene el archivo *Dockerfile* se necesita ejecutar el siguiente comando (ver Código 3.2.1.3) para generar la imagen y crear un contenedor en *Heroku*.

```
$ heroku container:push web
```

Código 3.2.1.3 Comando para crear contenedor en *Heroku*

En este capítulo se cubrieron las herramientas que se utilizaron para el desarrollo de software en equipo, desde comandos básicos a configuraciones intermedias de herramientas como *Docker Compose y Heroku.* Con el fin de que desde antes de empezar a programar la aplicación se tenga configurado el ambiente de desarrollo en todos los equipos de trabajo y con esto prevenir las demoras al momento de escoger las versiones de lenguaje de programación como del SMDB, etcétera.

En nuestro proyecto configurar el ambiente de desarrollo ayudó a agilizar el proceso tanto de repartición de tareas como de integración ya que lo podía hacer cada quién desde su equipo aunque esta configuración nos llevó más de un mes puesto que desconocíamos la mayor parte de estas herramientas y no se cubrían durante el curso.

Capítulo 4. Ejemplos de desarrollo

En este capítulo se muestra cómo se hizo el desarrollo de algunas secciones de la plataforma como el manejo de sesión, la generación y envío de archivos PDF[25].

4.1 Manejo de sesión con Devise

El manejo de sesión se puede hacer desde cero, aprendiendo así a profundidad el funcionamiento de *Ruby on Rails* aunque lleva más tiempo implementarlo.

En este documento se hace uso de la gema *Devise* por cuestiones didácticas ya que es una gema muy completa que incluye diez módulos:

- Autentificación por Base de Datos
- Bloqueo de cuentas
- Confirmación por correo
- Manejo de Cookies[26]
- Manejo de OmniAuth[27]
- Manejo de sesión por tiempo
- Rastreo
- Registro
- Recuperar contraseña
- Validaciones

Además de que permite generar el código de las vistas, controladores para modificarlo y así se adecua al proyecto en desarrollo.

Una vez agregado *Devise* al Gemfile se necesita ejecutar el comando *bundle install* para instalar la gema y ejecutar el comando (ver Código 4.1.1) que generará la configuración básica de *Devise*, además de que muestra en consola una serie de instrucciones que se tienen que seguir para su correcto funcionamiento.

```
$ rails generate devise:install
```

Código 4.1.1 Comando para iniciar configuración básica de gema *Devise*

[25] Formato de Documento Portátil, PDF por sus siglas en inglés *Portable Document Format*
[26] Información enviada por un sitio web y almacenada en el navegador del usuario
[27] Biblioteca que estandariza la autentificación en diferentes aplicaciones web

Una de las instrucciones es agregar la dirección web (URL) del servidor anfitrión en los archivos de configuración de ambiente, tanto en desarrollo como producción, para que el sistema de mensajería electrónica agregue un enlace hacia la dirección establecida, como se puede observar en la línea 3 del Código 4.1.3.

```
1. # Agregar en archivos config/environments/development.rb y
2. # config/environments/production.rb
3. config.action_mailer.default_url_options = { host: 'localhost', port:
3000 }
```

Código 4.1.2 Configuración de dirección web del servidor anfitrión

Al terminar la ejecución del comando se generan dos archivos, uno que es la configuración inicial de *Devise* y un archivo en donde están los mensajes que muestra cuando ocurre una acción, como confirmar el correo electrónico. Dentro del archivo de configuración *config/initializers/devise.rb* se tienen varias opciones preestablecidas que se pueden modificar como notificar a los usuarios mediante un correo electrónico cuando su cuenta haya cambiado de contraseña, poner un tiempo límite de inactividad, cuando haya transcurrido se le pedirá al usuario volver a iniciar sesión, etcétera[28].

Hasta este punto se tiene la configuración básica de *Devise,* pero no tiene ningún uso ya que no está asociada a un modelo. Para hacer la asociación se tiene que ejecutar el siguiente comando (ver Código 4.1.4 y Código 4.1.5).

```
$ rails generate devise Modelo
```

Código 4.1.4 Asociación de gema *Devise* con un modelo

```
$ rails generate devise User
```

Código 4.1.5 Asociación de gema *Devise* con model *User*

Cabe mencionar que si se requiere usar la gema *Devise* en más de un modelo, se debe de modificar el archivo *config/initializers/devise.rb* (ver Código 4.1.6) para poder tener las vistas y controladores por modelo. Si no se hace esta configuración todos los modelos van a compartir las mismas vistas y controladores.

[28] Para mayor información https://github.com/plataformatec/devise

```
# Archivo config/initializers/devise.rb
...
config.scoped_views = true
```

Código 4.1.6 Activar gema *Devise* en más de un modelo

Se recomienda revisar muy bien el archivo *config/initializers/devise.rb* ya que dentro de éste se pueden realizar configuraciones bastante útiles como cambiar el tiempo que tiene un usuario para activar su cuenta mediante el enlace que recibe por correo electrónico, el tiempo que puede durar la sesión del usuario antes de que tenga que introducir nuevamente sus credenciales, las credenciales con las que se puede iniciar sesión ya sea correo, nombre de usuario, etcétera.

Para modificar los estilos de las vistas o el funcionamiento de los controladores de los módulos de *Devise* (ver Código 4.1.7) se tienen dos opciones, generar las vistas o controladores de todos los módulos (ver Código 4.1.8) o sólo las vistas o controladores de los módulos que se quieran modificar (ver Código 4.1.9 y Código 4.1.10).

```
$ rails generate devise:[views,controllers] [Modelo]
```

Código 4.1.7 Comando para generar vistas/controladores de módulos

```
$ rails generate devise:views users
```

Código 4.1.8 Comando para generar todas las vistas asociadas a *User*

```
$ rails generate devise:views users -v confirmations mailer registrations passwords sessions unlocks shared
```

Código 4.1.9 Comando para generar ciertas vistas asociadas a *User*

```
$ rails generate devise:controllers users -c confirmations mailer registrations passwords sessions unlocks shared
```

Código 4.1.10 Comando para generar ciertos controladores asociados a *User*

4.1.1 Registro de usuario

Como se ve en la Figura 4.1.1.1 la página que se crea es básica tanto en campos como en estilos, sólo consta de correo electrónico y contraseña.

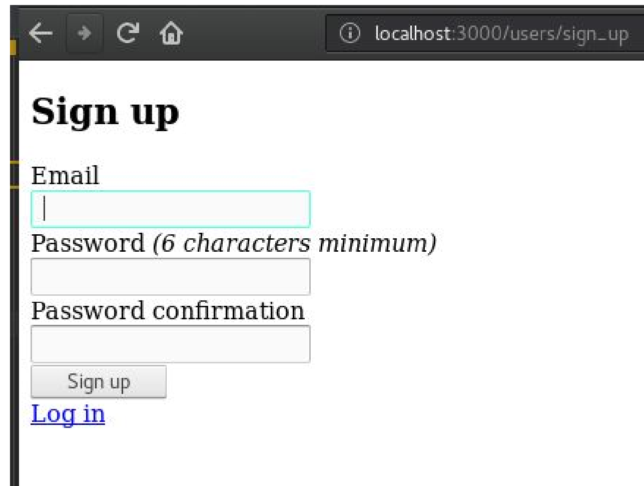

Figura 4.1.1.1 Registro predeterminado de *Devise*

Sin embargo esta gema es flexible, por lo que permite realizar modificaciones para agregar cuantos campos sean necesarios y modificar los estilos, ver Figura 4.1.1.2. Para modificar el registro se tienen que generar la vista y el controlador, para esto se ejecutan los comandos vistos en el Código 4.1.9 y Código 4.1.10 usando el módulo *registrations*.

Figura 4.1.1.2 Registro modificado de *Devise* con estilos

4.2 Creación de recursos de manera asíncrona

A continuación se muestra cómo se pueden crear recursos ya sea que se requiera tener una sesión activa o no. Se mostrará el desarrollo de dos casos, uno de comentarios que es el tipo de recurso que cualquiera puede crear y el otro es un *CRUD* (*Create, Read, Update, Delete; que en español se traduce como Crear, Leer, Actualizar y Borrar*) de cursos que sólo un usuario activo y que sea de tipo profesor puede ejecutar.

Para los siguientes ejemplos se hará uso de *AJAX*[29] con el formato *JSON*[30] para tener un sistema que pueda ser usado en cualquier tipo de aplicación ya sea web, de escritorio y/o móvil. Hacerlo con *HTML* y *Ruby* también es correcto si se piensa utilizar para un proyecto al que sólo se acceda desde un navegador web pero si se piensa continuar con el proyecto y extenderlo a dispositivos móviles es muy recomendable desde un inicio utilizar *JSON* para el intercambio de datos y seguir un estilo arquitectónico como *REST*[31] o utilizar un lenguaje de consultas para el lado del cliente como *GrahpQL*[32] que permita manejar toda la información desde una sola ruta.

4.2.1 Recursos con sesión obligatoria

Siguiendo el planteamiento de la sección anterior primero se hará el diseño de la base de datos, después se pasará eso a migraciones y modelos, y por último se escribirá el controlador. Como se mencionó anteriormente en esta sección se manejará un CRUD de cursos con la restricción que sólo los usuarios que sean profesores puedan hacer uso de éste. En la Figura 4.2.1.1 se muestra el modelo Entidad-Relación que se va a utilizar durante esta sección, en ésta se muestra la información que guardan los profesores y los cursos así como que los cursos dependen de los profesores.

[29] Técnica que evita recargar la página para obtener información, haciendo que sea más interactiva. JavaScript asíncrono y XML, Asynchronous JavaScript And XML. https://skillcrush.com/2018/04/26/what-is-ajax/

[30] Formato de intercambio de datos. Notación de Objetos de JavaScript, JavaScript Object Notation https://tools.ietf.org/html/rfc7159

[31] Estilo de arquitectura para diseñar aplicaciones en red, se usa en sistemas que utilicen HTTP para obtener datos o manipular esos datos, por lo regular se utilizan formatos como XML y JSON. https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest

[32] Lenguaje de consulta y manipulación de datos que utiliza se utiliza desde el lado del cliente. https://graphql.org/

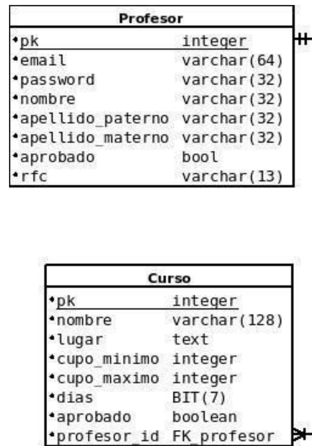

Figura 4.2.1.1 Diagrama Entidad-Relación de profesores y cursos

A continuación se definen las migraciones de los modelos, donde sólo es pasar el diagrama a código que entienda *Ruby on Rails* como se aprecia en el Código 4.2.1.1 en el que se agregan los mismos campos que se tienen en la Figura 4.2.1.1.

```
# db/migrate/create_professors.rb
class CreateProfessors < ActiveRecord::Migration
 def change
   # Se agrega la tabla de profesores junto con los campos que contiene
   create_table :professors do |t|
     t.string    :email,                null: false,limit: 64
     t.string    :password,             null: false,limit: 32
     t.string    :nombre,               null: false,limit: 32
     t.string    :apellido_paterno,     null: false,limit: 32
     t.string    :apellido_materno,     null: false,limit: 32
     t.boolean   :aprobado,             null: false,default: false
     t.string    :rfc,                  null: false,limit: 13
     t.timestamps null: false
   end
   # Se agrega un índice al campo de email para aumentar la velocidad de
   # búsqueda por éste campo
   add_index :email,                 unique: true
 end
end
```

Código 4.2.1.1 Archivo con la migración de los profesores

```
# db/migrate/create_courses.rb
class CreateCourses < ActiveRecord::Migration
 def change
   # Se agrega la tabla de cursos junto con los campos que contiene
   create_table :courses do |t|
     t.string     :nombre,                 null: false,limit: 128
     t.text       :lugar,                  null: false,limit: 32
     t.integer    :cupo_minimo,            null: false,default: 1
     t.integer    :cupo_maximo,            null: false,default: 2
     t.boolean    :aprobado,               null: false,default: false
     t.column     :dias,                   "BIT(7)",null: false
     # Se agrega una relación de pertenencia a la tabla de profesores
     t.belongs_to :professor,              index: true
     t.timestamps null: false
   end
 end
end
```

Código 4.2.1.2 Archivo con la migración de los cursos

Ya que se tienen las migraciones se definen las relaciones en la sección *models* para poder hacer referencia a éstas de la misma forma en la que se manipulan los otros campos, en el modelo de profesores (ver Código 4.2.1.3) se especifica que un profesor puede tener muchos cursos.

```
# models/professor.rb
class Professor < ActiveRecord::Base
 has_many :courses
end
```

Código 4.2.1.3 Configuración de modelo de profesores

En el modelo de cursos (ver Código 4.2.1.4) se agregan funciones de filtrado que evitan la repetición de código.

```ruby
# models/course.rb
class Course < ActiveRecord::Base
 belongs_to :professor

 def self.aprobados
   where(aprobado: true)
 end

 def self.search(course_name)
   where('nombre ILIKE ? and aprobado = true', "%#{course_name}%")
 end
end
```

Código 4.2.1.4 Configuración de modelo de cursos

Una vez definidos los modelos se escribe el controlador ya que así ya se sabe los campos con los que se va a trabajar, en el controlador se va a habilitar una función la cual no va a necesitar tener una sesión activa para que cualquier usuario pueda ver los cursos disponibles, así como funciones para editar, borrar y crear cursos que necesitan de una sesión activa (ver Código 4.2.1.5).

```ruby
# controllers/courses_controller.rb
class CourseController < ApplicationController
 # 'before_action' es una función que se aplica antes de ejecutar el
código
 # de las funciones especificadas

 # Verifica que el profesor tenga una sesión activa para poder
 # crear, actualizar o eliminar
 before_action :is_logged?, only: %i[create update destroy]
 # Verifica que el usuario tenga permisos sobre el curso, en este caso
 # sólo necesita ser profesor y estar asociado al curso
 before_action :is_owner?, only: %i[update destroy]
 # Lee el parámetro id de la petición y busca el curso correspondiente
 before_action :set_course, only: %i[show update destroy]
 # GET courses?search=course_name
```

```ruby
def index
  # Si se pasa el parámetro search se realiza una búsqueda bajo el
  # campo nombre sino se obtienen todos los cursos disponibles que se
  # encuentran activos
  @courses = if params.has_key?(:search) then
               Course.aprobados
             else
               Course.search(params[:search])
             end
  render json: { courses: @courses }
end

# GET courses/:id.json
# GET courses/:id.pdf
def show
  # Se obtiene la información del profesor asociado al curso y la
  # información del curso, la respuesta puede ser un JSON o un PDF
  @professor = @course.professor
  respond_to do |format|
    format.json do
      render json: { course: @course, professor: @professor }, status:
:ok
    end
    format.pdf do
      render pdf: @course.nombre, disposition: 'attachment'
    end
  end
end
# POST courses
def create
  # Obtiene el profesor activo y le asocia el curso
  professor = current_professor
  course = professor.courses.build(course_params)
```

```
    # Agrega los días como bits para no ocupar mucho espacio en la base
    course.dias = map_days(course_params[:dias])
    # Verifica si el curso se puede crear y guardar, manda el mensaje
    # pertinente en cada caso
    if curso.valid?
      if curso.save!
        render json: { message: 'Course created' }, status: :created
      else
        render json: course.errors, status: :unprocessable_entity
      end
    else
     render json: course.errors, status: :bad_request
    end
  end
  # PUT courses/:id
  def update
    # Actualiza la información del curso especificado
    new_days = map_days(course_params[:dias])
    if @course.update_attributes(curso_params)
      @course.update(dias: new_days)
      render json: { message: 'Course updated',course: @course }, status:
:ok
    else
      render json: course.errors, status: :unprocessable_entity
    end
  end

  # DELETE courses/:id
  def destroy
    # Elimina el curso previamente asignado y redirige al usuario a la
    # dirección indicada como raíz
    @curso.destroy
```

```
  render json: { message: 'Course deleted' }, status: :no_content
 end
 private
 # Asocia los días de la semana con su nombre en español
 def map_days(course_days)
   days = ['lunes', 'martes', 'miercoles', 'jueves',
           'viernes', 'sabado', 'domingo']
   day_bits = course_days.map { |day| (days.include? day.downcase) ? 1 :
0 }
   day_bits.join('')
 end
 # Checa que se reciba el parámetro course de manera obligatoria y de
 # manera opcional los campos nombre, lugar, cupo_min, cupo_max
 def course_params
   params.require(:course).permit(:nombre, :lugar, :cupo_min, :cupo_max,
                                  dias: [])
 end
 def is_logged?
   # Verifica que el usuario tenga una sesión activa sino manda un
   # código de estado 401 - Unauthorized
    render json: { message: 'User anonymous is not permitted' }, status:
:unauthorized unless current_professor
  end

 def is_owner?
   # Verifica que el usuario sea el dueño del curso sino manda un
   # código de estado 401 - Unauthorized
     render json: { message: 'User has not permitted this action' },
status: :unauthorized unless
Course.exists?(:professor_id=>current_professor.id)
 end

 def set_course
```

```
  @course = Course.find(params[:id])
 end
end
```

Código 4.2.1.5 Configuración de controlador de cursos

4.3 Generación de archivos PDF

Para la generación de archivos se utiliza la gema *WickedPDF* que permite crear archivos *PDF* a partir de plantillas *HTML* para facilitar el diseño de los archivos, la cual usa la aplicación *wkhtmltopdf*[33] para convertir los archivos *HTML* a *PDF*, se puede instalar desde el sistema operativo o utilizando la gema *wkhtmltopdf-binary*.

Después de agregar las gemas al archivo Gemfile se ejecuta el comando *bundle install* para que se realice la instalación. Una vez finalizada la instalación se necesita ejecutar el siguiente comando (ver Código 4.3.1) para que se genere el inicializador *wicked_pdf.*

```
$ rails generate wicked_pdf
```

Código 4.3.1 Inicializar gema *WickedPDF*

En algunas versiones de rails se necesita agregar el tipo *pdf* al inicializador de MIME Type esto con el fin de poder usarlo como tipo de respuesta.

```
# config/initializers/mime_types.rb
Mime::Type.register "application/pdf", :pdf
```

Código 4.3.2 Agregar *PDF* a tipo de respuesta

Ya que se tiene la configuración inicial sólo falta diseñar el *HTML* que se va a utilizar como plantilla y modificar el controlador para devolver el archivo *PDF*. Como ejemplo se muestra un plantilla *HTML* que acomoda los comentarios en una tabla con el nombre de usuario y la descripción del comentario.

```
<!DOCTYPE html>
<html>
<head>
```

[33] http://wkhtmltopdf.org

```
<meta charset="utf-8">
<style type="text/css">
 table, td, th {
  border: 1px solid black;
 }
 table {
  border-collapse: collapse;
  width: 100%;
 }
</style>
</head>
<body>
<table>
 <thead>
 <tr>
   <th colspan="3">Comentarios del día</th>
 </tr>
 </thead>

 <tbody>
 <tr>
   <th>Usuario</th>
   <th>Descripción</th>
 </tr>
 <% @comments.each do |comment| %>
   <tr>
     <td><%= comment.username %></td>
     <td><%= comment.text %></td>
   </tr>
 <% end %>
 </tbody>
</table>
</body>
</html>
```

Código 4.3.3 Plantilla *HTML* de comentarios de usuario

Como se aprecia en el Código 4.3.3, estos archivos también hacen uso de *ERB*[34] por lo que hace más fáciles ciertas cosas, en este caso crear una tabla a partir de una lista de comentarios, la cual se genera desde el controlador. A continuación se muestra el código del controlador que genera esa lista además de cómo se especifica que únicamente debe responder a las peticiones de *JSON* y *PDF*.

```
# controllers/articles_controller.rb
...
# GET /articles/:id/comments.json
# GET /articles/:id/comments.pdf
def show
 # Obtiene todos los comentarios asociados al artículo
 @comments = @article.comments
 # Se validan los tipos de respuesta, en este caso sólo responde cuando
 # se solicitan JSON o el archivo PDF
 respond_to do |format|
   format.json do
     render json: { comments: @comments }, status: :ok
   end
   format.pdf do
     render pdf: @article.title
   end
 end
end
```

Código 4.3.4 Código para responder con archivo *PDF*

Tal cual como está el código se abrirá el archivo *PDF* en la misma ventana que se realiza la petición y el nombre del archivo va a ser el título del artículo, si se requiere que aparezca la ventana para descargar el archivo (Ver figura 4.3.1) se necesita especificar dentro de la instrucción *render*.

[34] Embedded Ruby, es una característica que permite incrustar código de *Ruby* en plantillas, ya sean HTML, texto plano, JavaScript, etcétera.

```ruby
format.pdf do
     render pdf: @article.title, disposition: 'attachment'
end
```

Código 4.3.5 Código para que se descargue archivo *PDF*

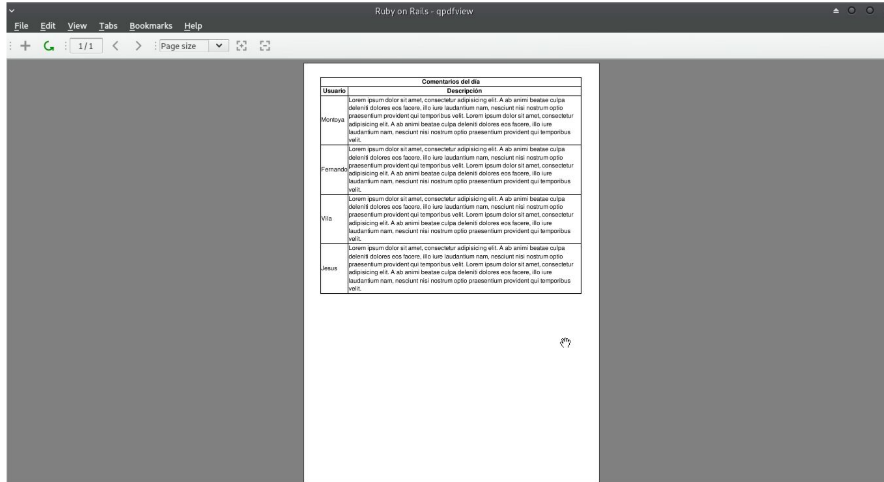

Figura 4.3.1 *PDF* generado a partir de plantilla *HTML*

Realizar el desarrollo con un lenguaje como *Ruby* tiene su complejidad cuando no se conoce la sintaxis pero es de gran ayuda para desarrollar un proyecto en poco tiempo por la gran cantidad de información y ejemplos que se encuentran en la web, además de que tiene varias gemas que facilitan algunas tareas como *Ruby on Rails* que genera la configuración básica de una aplicación web.

Conclusiones

El objetivo principal de este manual fue presentar una forma en la que se pueda trabajar en equipo de forma sencilla y eficiente con *Ruby on Rails* y otras tecnologías como *Git, Docker y Heroku*. Esto es importante para los cursos de Ingeniería de Software de la Facultad de Ciencias de la Universidad Nacional Autónoma de México, también se puede utilizar para otros cursos o durante la etapa laboral.

Al principio de este manual se expusieron conceptos básicos de las metodologías que se utilizaron durante el desarrollo del sistema como *SCRUM, Trello*, etcétera. Que se utilizan para planear el proyecto y organizar el trabajo del equipo de desarrolladores.

En este trabajo se expusieron conceptos básicos de *Ruby on Rails*, así como algunos ejemplos de los comandos más utilizados de las tecnologías que se manejan durante el desarrollo de software en equipo.

La parte más importante de este documento fue el presentar un ambiente de desarrollo que agilice el desarrollo entre varias personas y disminuir los tiempos de desarrollo, así como los costos monetarios en un ámbito laboral.

Por último, se mostraron algunos ejemplos que puedan ser de gran ayuda para quien desee hacer un proyecto web ya que incluye el manejo de la gema *Devise* y puesto que la mayoría de los sistemas requieren un sistema de manejo de sesión, esta gema resulta bastante robusta.

Si bien el desarrollo del proyecto es una parte esencial para el cliente pues éste puede visualizar y utilizar las pantallas y vistas de proyecto, se debe tener en cuenta que la parte de diseño y documentación deben de haberse generado antes de empezar el desarrollo ya que tener por escrito las especificaciones del sistema ayuda a saber por dónde comenzar y qué es lo que el cliente quiere por eso se recomienda revisar el documento *CREACIÓN Y DOCUMENTACIÓN DE UN SISTEMA WEB PARA LA SECRETARÍA DE EDUCACIÓN ABIERTA Y CONTINUA* [5] que es complemento de este trabajo.

Bibliografía

[1] Álvarez, G., Lasa, C., & De las Heras, R. 352p. (2012). *Métodos Ágiles y Scrum*. Madrid, España: Anaya Multimedia.

[2] Anderson J. 262p. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*. -: Blue Hole Press.

[3] Cedillo, A. 250p. (2015). *Material de Apoyo para el laboratorio de Ingeniería de Software usando el marco de trabajo Ruby on Rails. Trabajo de apoyo a la docencia para titularse de licenciatura en Ciencias de la Computación*. Facultad de Ciencias, UNAM:.

[4] Chacon, S., & Straub, B. 517p. (2014). *Pro Git.* -: Apress.

[5] Sánchez, F. 153p. (2018). *Creación y Documentación de un Sistema Web para la Secretaría de Educación Abierta y Continua. Trabajo de apoyo a la docencia de la licenciatura en Ciencias de la Computación.* Facultad de Ciencias, UNAM:.