UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Diseño e implementación del sistema PEAT
(Progressive Energy Audit Tool)

REPORTE DE TRABAJO PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Héctor Enrique Gómez Morales

TUTORA

M. en I. Karla Ramírez Pulido

2016

Hoja de Datos del Jurado

1. Datos del alumno
Gómez
Morales
Héctor Enrique
5518505001
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
401048742

2. Datos del tutor
M en I
Karla
Ramírez
Pulido

3. Datos del sinodal 1
Dra
María de Luz
Gasca
Soto

4. Datos del sinodal 2
Dr
Daniel
Trejo
Medina

5. Datos del sinodal 3
Dr
Jorge Luis
Ortega
Arjona

6. Datos del sinodal 4
Act
Carlos Ernesto
López
Natarén

7. Datos del trabajo escrito
Diseño e implementación del sistema PEAT (Progressive Energy Audit Tool)
100 p
2016

Agradecimientos

Agradezco el inagotable amor y apoyo de mis padres, sin ellos este trabajo nunca hubiera sido posible.

A mis hermanos Javier y Lilian que durante muchos años fueron mis segundos padres y de los cuales he aprendido tanto.

A mi tutora Karla Ramírez Pulido le tengo un eterno agradecimiento por la constante ayuda y asesoría dada no solamente para la elaboración de este reporte si no durante mi paso por la facultad. También agradezco la oportunidad de haber sido su ayudante en uno de mis cursos favoritos que es Lenguajes de Programación.

Agradezco a mis sinodales Dr. Maria de Luz Gasca Soto, Dr. Daniel Trejo Medina, Dr Jorge Luis Ortega Arjona y al Act. Carlos Ernesto López Natarén por los comentarios realizados ya que éstos enriquecieron mi trabajo.

A mis amigos David García Gudiño, Víctor Hugo Ortiz Hernández, Pavel David Sulca Cavero y María Margarita López Titla por su apoyo y amistad que ha resistido el tiempo y distancia.

También agradezco la amistad y apoyo de mis colegas Juan Germán Castañeda Echevarría, Jesús Alejandro Juárez Robles y César Octavio López Natarén por su gran pasion y profesionalismo por nuestra profesión que ha sido una gran inspiración para mi.

Agradezco a la Facultad de Ciencias y a la Universidad Nacional Autónoma de México por darme acceso a tan excelentes profesores que me han formado.

Índice general

Introducción

El presente trabajo aborda el diseño e implementación del sistema *Progressive Energy Audit Tool* (PEAT), el cual es un sistema web que permite a los usuarios de pequeñas y medianas empresas (PyMES) identificar y monitorear sus gastos de energía eléctrica, con el fin de obtener un mayor control en los costos de ésta.

Este trabajo se realizó siendo parte de la empresa C3 Energy en el cargo de Senior Software Engineer por un período de 10 meses, de Mayo de 2012 a Febrero de 2013.

En una gran cantidad de proyectos resulta ser suficiente el hacer uso de un único lenguaje de programación para llevar a buen término el proyecto, es decir, terminarlo de forma exitosa. Sin embargo conforme pasa el tiempo, las necesidades de las compañías y usuarios cambian, lo cual provoca que hagamos uso de una serie de tecnologías no previstas en un inicio, lo que representa casi siempre el uso de más de un lenguaje de programación[1].

El proyecto de implementación del sistema PEAT cuenta con tres partes principales de desarrollo: (a) el *backend* implementado en su mayoría usando el lenguaje de programación Javascript, con un núcleo escrito en el lenguaje de programación Java, (b) una biblioteca en el lenguaje de programación Ruby que permite la interacción del *backend* con aplicaciones web y (c) el *frontend* escrito con Ruby, dada sus ventajas para realizar sistemas web.

Durante este proyecto se implementó una biblioteca no solo para hacer uso de los servicios web ya existentes, sino para facilitar la creación de clases y objetos en Ruby basados en la jerarquía de clases definida en Javascript. Cabe mencionar que haciendo uso de una de las fortalezas de Ruby como lo es la metaprogramación para la creación de clases y sus atributos "al vuelo", según las especificaciones enviadas por el servidor, se mantiene así la jerarquía de clases de un lenguaje a otro.

Otro punto de creciente importancia es el despliegue de una aplicación, ya que por un lado hace algunos años las aplicaciones web se desarrollaban bajo el modelo de tres capas: la base de datos, el servidor de la aplicación y el servidor web; actualmente se tienen más servicios que deben estar en línea sobre todo para garantizar un servicio concurrente; repositorios de memoria, balanceadores de carga, servidores de cola, etcétera. Por lo que automatizar el despliegue de una aplicación en sus diferentes contextos, desarrollo, producción y pruebas, es de vital importancia para el desarrollo de software en tiempo y en forma.

[1] Debido a que ningún lenguaje de programación actualmente es el mejor en todos los contextos posibles de uso.

Durante este proyecto se hace uso extensivo de técnicas como la metaprogramación y el despliegue continuo, para permitir un desarrollo acelerado, con el fin de obtener rápidamente retroalimentación del usuario final.

Objetivo general

El objetivo general del sistema PEAT es dar la mayor utilidad posible a usuarios PyMES con la menor información disponible, fomentando en el usuario el compartir más información sobre su empresa, obteniendo un mejor control acerca de su consumo energético.

El sistema debe además permitir el ingreso progresivo de información, por medio de una serie de preguntas específicas al usuario, dando mejores recomendaciones para bajar su consumo energético conforme el sistema obtiene mas información.

A partir de la información obtenida el sistema debe permitir el monitoreo y revisión del consumo energético de forma detallada, ya sea en horas, días, meses y años.

Objetivos secundarios

Los objetivos secundarios que apoyan al objetivo general de este trabajo son:

- Implementación de una interfaz que permita la obtención de información del usuario de una forma eficaz y sencilla.
- Dar información útil aunque el usuario solo proporcione el mínimo de información sobre su empresa.
- Proporcionar recomendaciones para disminuir sus gastos en energía con base en el consumo e información proporcionada hasta el momento.
- Autentificar a los usuarios mediante el uso de credenciales de acceso obtenidas en el portal web de Pacific Gas and Electric Company (PG&E).
- Diseñar e implementar un conjunto de pruebas unitarias, funcionales y de integración para los módulos críticos del sistema.
- Soportar por lo menos a mil usuarios concurrentes.
- Implementar la infraestructura para el despliegue continuo de la aplicación, permitiendo una retroalimentación contínua sobre el funcionamiento del sistema.

Organización del trabajo

Este trabajo está dividido en cuatro capítulos los cuales son:

- Capítulo 1 titulado, *Situación original*: se presenta una descripción del contexto que dio pie al desarrollo del sistema PEAT, después se da una descripción de la arquitectura y sistemas con los que se tuvo como punto de partida para su implementación. Finalmente se define la propuesta de desarrollo del sistema.

- Capítulo 2 que lleva por nombre, *Fundamentos teóricos*: se expone un resumen de los conceptos teóricos mas influyentes en el diseño e implementación del sistema PEAT.
- Capítulo 3, *Diseño e implementación del sistema PEAT*: se desarrolla y describe la implementación del sistema PEAT, dando la justificación de las decisiones tomadas durante su implementación.
- Capítulo 4, *Despliegue del sistema*: se expone la infraestructura del ambiente de producción y la implementación de un proceso de despliegue continuo del sistema PEAT y las decisiones tomadas para obtener un rendimiento óptimo del sistema.

El sistema PEAT es resultado del trabajo en conjunto de tres compañías: Pacific Gas and Electric Company (PG&E), C3 Energy y Software Next Door. Cabe mencionar que PG&E es una compañía proveedora de gas natural y electricidad, una de las más grandes compañías de Estados Unidos con sede en San Francisco, California. El sistema PEAT es el resultado de una licitación iniciada por PG&E, siendo ganadora de dicha licitación la compañía C3 Energy.

Capítulo 1

Situación original

En este capitulo se describe el contexto que dio pie al desarrollo del sistema PEAT, con una descripción de la arquitectura y sistemas ya existentes en la compañía C3 Energy, los cuales sirvieron como punto de partida para la implementación del sistema PEAT.

1.1. Contexto

De 2000 a 2002 se tuvo una crisis energética en el estado de California en los Estados Unidos de América, esta crisis provocó que el gobierno del estado empezara a tomar decisiones de largo plazo acerca de su estrategia energética. En los años subsecuentes se aprobaron nuevas leyes para incentivar el uso de energía renovable, la producción de energía, entre otros.

Para el 2010 se aprobó una nueva legislación que permitía un nuevo esquema de cobro a las empresas generadoras de electricidad. Este nuevo esquema llamado hora de consumo (*time-of-use*, TOU) define que las tasas de cobro varíen en función de la temporada y la hora del día en que se usa la energía eléctrica, en contraste con las tasas fijas tradicionales.

En un esquema TOU se definen tres tarifas y sus zonas de tiempo asociadas [19]:

- Demanda baja: es la tarifa más baja, se aplica durante las horas de la mañana, en la noche y los fines de semana.
- Demanda mediana: es la tarifa intermedia, se aplica durante las horas de 10 am a 1pm y de 7 pm a 9 pm.
- Demanda alta: es la tarifa más cara, se aplica durante las horas de 1 pm a 7 pm.

El objetivo de un esquema TOU es el distribuir la carga de la red eléctrica, alentando a los usuarios a cambiar su consumo durante períodos de demanda alta a períodos de demanda baja, al bajar la demanda en los períodos de demanda alta provoca que se reduzcan los costos de generación de energía de forma general.

Un esquema TOU es solo posible con la llegada de los medidores inteligentes, a diferencia de los analógicos los medidores inteligentes son capaces de medir el consumo de electricidad de forma instantánea, de distinguir y facturar el consumo de electricidad según el momento en que se esta realizando el consumo. Desde inicios del 2010 Pacific Gas and Electric Company (PG&E) empezó la instalación de medidores inteligentes en su territorio, por lo que para inicios del 2012 ya casi había completado la transición.

La fecha de arranque de la nueva legislación fue durante el mes de noviembre del 2012, mes en el cual las empresas como PG&E podían iniciar la transición a un esquema TOU de sus usuarios PyMES. Una parte vital que permitió la transición fue que la empresa PG&E brindará herramientas para que el usuario final pudiera analizar sus gastos de energía y así tomar decisiones sobre su consumo de energía según las nuevas tarifas y su historial de consumo.

Dada la situación descrita anteriormente es que nace el sistema *Progressive Energy Audit Tool* (PEAT) para solventar la necesidad de información del usuario final sobre sus gastos de energía en una forma detallada. La licitación para la implementación del sistema PEAT fue puesta a concurso por parte de PG&E siendo la compañía C3 Energy la ganadora de dicha licitación ya que contaba con la infraestructura necesaria para el procesamiento de una gran cantidad de datos de consumo de energía, además contaba con un sistema de monitoreo de consumo de energía enfocado a empresas de nivel multinacional. El reto era pasar de un sistema y procesos diseñados para una docena de clientes, a un sistema que diera servicio a cientos de miles de clientes PyMES.

Para el funcionamiento del sistema PEAT es necesario contar con una cantidad considerable de información del consumo de energía del usuario por lo que el sistema esta enfocado a PyMES que cuenten con medidores inteligentes y con un historial de consumo de por lo menos un año.

1.2. Arquitectura previa

En C3 Energy ya se contaba con toda una infraestructura para el procesamiento y análisis en tiempo real de datos de consumo eléctrico y de gas.

El sistema consta de cuatro capas:

- Una capa de almacenamiento de información que era de referencia o datos ya procesados
- Una capa de análisis en las que se hacía el análisis de los datos y que también tomaba el rol de la capa de caché.
- Una capa de servicios web que permitía acceder a los datos contenidos en las capas anteriores.
- Una capa en donde se encuentran los sistemas que hacen uso de los servicios web para dar información al usuario final.

1.2.1. Bases de datos

Las dos primeras capas almacenan y analizan la información obtenida de un gran número de bases de datos externas relacionadas con temas de energía. Todos estos datos son concentrados en una base de datos central en la que se usaba Oracle Database 10g[1] para este rol.

También se contaba con un *cluster* de Apache Cassandra[2] el cual era usado para realizar el análisis en tiempo real de los datos, además de servir como capa de caché para los datos mas solicitados en el sistema.

1.2.2. Servidor y API

Sobre la base de datos anterior se tenía un API[3] la cual permitía el hacer consultas y operaciones sobre los datos contenidos y/o analizados en este sistema. Este API estaba implementado en Javascript bajo Rhino[4] mismo que se ejecuta bajo la máquina virtual de Java (*Java Virtual Machine*, JVM), obteniendo como ventaja principal el poder realizar la implementación de rutinas críticas en un lenguaje con mejor rendimiento como lo es Java. De esta forma se tenía el núcleo del servidor implementado en Java con el resto del API implementada en Javascript.

1.2.3. Sistemas en producción

Los sistemas existentes hasta ese momento (2012) para varias empresas multinacionales eran del tipo SPA (*Single Page Application*) los cuales son conocidos como sistemas web que tienen el fin de dar una experiencia de usuario similar a la de una aplicación de escritorio [16].

Estos sistemas fueron implementados haciendo uso del marco de trabajo desarrollado dentro de la compañía usando Javascript. Se tenía una gran jerarquía de clases la cual era compartida tanto con el servidor como con el cliente, obteniendo una sola fuente de la descripción de las clases y objetos del sistema.

Se utilizaba el marco de trabajo Ext JS, enfocado a construir sistemas SPA en Javascript, usado principalmente por la facilidad para generar todo tipo de gráficas rápidamente y de esta forma mostrar una gran cantidad de información al usuario.

Por lo descrito en los párrafos anteriores podemos decir que tanto en el *frontend* como en el *backend* hacían uso del lenguaje de programación Javascript principalmente con un núcleo de Java para las partes que requerían un rendimiento óptimo con el fin de manejar peticiones concurrentes, por lo que las ventajas al usarlo son:

- Reducir al mínimo el cambio de contexto de lenguajes entre el *frontend* y el *backend*.

[1] Es un sistema de gestión de base de datos de tipo objeto-relacional desarrollado por Oracle.

[2] Es una base de datos de tipo NoSQL distribuida, basada en un modelo de almacenamiento llave-valor, es desarrollada por Apache Software Foundation [9].

[3] *Application Programming Interface* es el conjunto de procedimientos que ofrece una biblioteca para ser utilizado por otro software [23].

[4] Un intérprete de Javascript de código abierto desarrollado en Java [17].

- Reducir sustancialmente la cantidad de código redundante al no tener que re-implementar la jerarquía de objetos, que era enviada por el servidor en formato JSON e interpretada por el cliente.

- Existía una única representación de una clase[5] por ejemplo la clase *Building* era una clase de Javascript que se usaba tanto en el servidor como en el cliente.

1.3. Propuesta de desarrollo

La primera propuesta para el desarrollo de PEAT consistía en implementar una aplicación de tipo SPA (como los otros sistemas existentes dentro de C3 Energy), es decir, desarrollar un cliente de Javascript que hiciera uso del API y jerarquía de objetos ya existente (con ciertas adicciones y modificaciones). Los primeros prototipos presentaban varios problemas bajo este esquema:

- El tiempo para realizar la precarga de todos los módulos y objetos dados por el API en el cliente tomaba un tiempo considerable (5+ segundos). Los sistemas anteriores eran usados por un número reducido de personas autorizadas, que hacían uso extensivo del sistema en sesiones de larga duración, por lo que este tiempo de arranque era tolerable.

- El tipo de interfases y la experiencia de usuario que se obtenían usando Ext JS eran excelentes en ese contexto, pues mostraban una gran cantidad de información al usuario, por otro lado en PEAT el volumen de información era menor y el aspecto de mayor relevancia era el mostrar al usuario dicha información de manera más fluída y rápida.

Para solucionar los problemas antes mencionados, se tomó la decisión de usar otro tipo de herramientas para construir aplicaciones finales al usuario, que permitieran reusar todo el API y la jerarquía de objetos implementados hasta ese momento.

Después de desarrollar varios prototipos con diferentes tecnologías se decidió usar el lenguaje de programación Ruby y el marco de trabajo Ruby on Rails (Rails), las razones principales fueron:

- Ruby tiene un gran soporte para la metaprogramación, es decir, para la creación de clases y sus atributos “al vuelo”, lo cual es de vital importancia para lograr una buena integración con la jerarquía de clases ya definida en el *backend*.

- Ruby tiene una gran capacidad para implementar lenguajes de dominio específico (*Domain Specific Language*, DSL), los cuales son lenguajes diseñados para solucionar problemas de un dominio en particular, característica que será usada para lograr la integración con el *backend* y la automatización del despliegue de la aplicación.

[5] En Javascript (ECMAScript 5) no hay clases pero dentro del marco de trabajo interno se definen por medio de funciones de construcción y herencia de prototipo.

- Gran integración con una gran cantidad de herramientas y bibliotecas para desarrollar, diseñar e implementar interfases de usuario.
- Gran soporte para realizar extensas pruebas unitarias, funcionales y de integración, por medio de bibliotecas como *RSpec*, *Cucumber*, *Capybara*, etcétera.

En el proyecto de implementación del sistema PEAT cuenta con tres partes principales de desarrollo: (a) el *backend* (Figura 1.1 - A) implementado en su mayoría usando Javascript, con un núcleo escrito en Java, (b) Bezel (Figura 1.1 - B) una biblioteca en Ruby que permite la interacción del *backend* con el servidor de PEAT y (c) el *frontend* (Figura 1.1 - C) escrito también con Ruby haciendo uso del marco de trabajo Ruby on Rails (Rails).

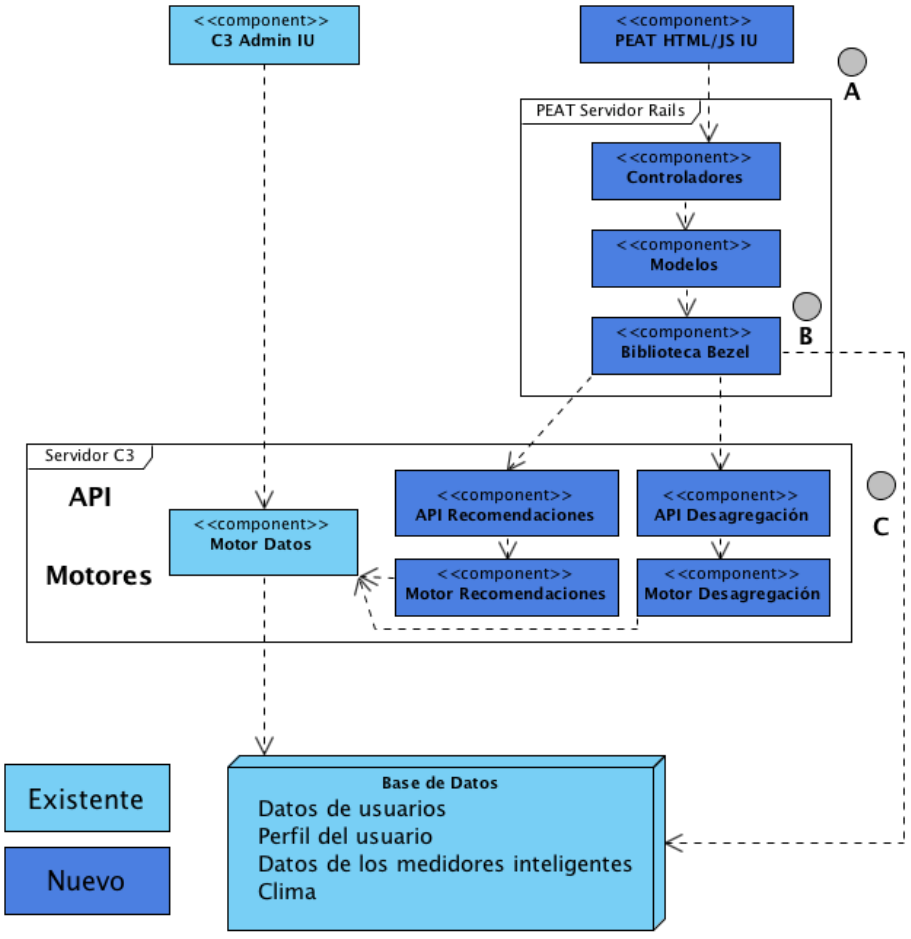

Figura 1.1: Esquema de alto nivel de la arquitectura para el sistema PEAT

El *frontend* está implementado por medio del marco de trabajo Rails, haciendo uso de la biblioteca *Backbone* para dar estructura a los módulos de Javascript que se utilizan en la vista. Para las gráficas y manejo de datos se seguirá haciendo uso de Ext JS por su gran facilidad para implementar cualquier tipo de gráficas.

El servidor de Rails recibe las peticiones de la interfaz web y coordina las peticiones necesarias al *backend*. El trabajo de la realización de las peticiones a los servicios web ya existentes para la creación de clases y objetos en Ruby basados en la jerarquía de clases definida en Javascript es realizada por la biblioteca `Bezel`, la cual es implementada en su totalidad.

Aunque en la parte del *backend* tiene un sistema de procesamiento y análisis de datos, es necesario diseñar e implementar dos nuevos componentes en éste. El primero es el módulo de recomendaciones (*Recommendations API*), el cual según las respuestas obtenidas por el usuario se utilizan para generar recomendaciones hechas a la medida del contexto de operación de cada usuario. El segundo módulo es el de desagregación (*Disaggregation API*) que toma los datos de consumo del usuario y usando aprendizaje de máquina separa dicho consumo en sus diferentes partes.

Para acelerar el desarrollo del sistema se implementó el *frontend* y el *backend* en forma concurrente y dado que una parte significativa estaría en desarrollo, se optó por usar datos estáticos contenidos en una base de datos como PostgreSQL, mientras se esperaba la implementación de los nuevos componentes del *backend* y de la implementación de la biblioteca `Bezel`.

Dado que al principio el *frontend* haría uso de un *backend* provisional con datos estáticos, al hacer el cambio al *backend* final se tendrían varias incompatibilidades, para acelerar este proceso de integración se propuso el uso de un conjunto de pruebas de integración utilizadas principalmente para asegurar que la transición fuera exitosa y con la menor cantidad de errores.

Como se puede ver en este capitulo el sistema `PEAT` no solo es parte de un conjunto ya existente de sistemas en operación si no que hace uso de estos sistemas para dar un servicio de suma importancia a una audiencia de tamaño considerable.

Capítulo 2

Fundamentos teóricos

En este capitulo se presentan los conceptos teóricos que permitieron el diseño e implementación del sistema PEAT.

2.1. Patrón Modelo-Vista-Controlador (MVC)

El patrón Modelo Vista Controlador (MVC) es probablemente el patrón mas utilizado y citado para el desarrollo de interfaces de usuario y sistemas web. MVC consiste de tres tipos de objetos [11]:

- Modelo: representación de la información de dominio del sistema.
- Vista: representación visual del modelo.
- Controlador: define la forma en que la interfaz reacciona a la entrada del usuario.

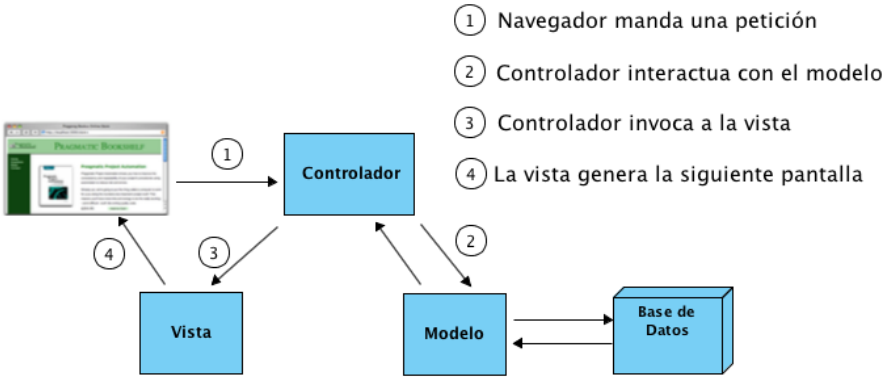

Figura 2.1: Patrón MVC para sistemas web [22].

MVC fue ideado originalmente para aplicaciones gráficas convencionales, donde los desarrolladores encontraron que la separación de responsabilidades, entre la presentación (vista y controlador) y el dominio (modelo), fomentadas por el patrón llevan a un menor acoplamiento lo que con lleva una escritura de código fuente y mantenimiento del mismo mucho mas sencillas (Ver Figura 2.1).

MVC desacopla vistas y modelos mediante el establecimiento de un protocolo de suscripción / notificación. La vista debe asegurar que el aspecto visual refleje el estado del modelo. Cada vez que cambian los datos del modelo, el modelo notifica a las vistas que dependen de ella. Este enfoque permite conectar múltiples vistas a un modelo para proporcionar diferentes presentaciones. También puede crear nuevas vistas para un modelo sin reescribir este último [12].

2.1.1. MVC y Rails

El marco de trabajo Rails hace uso de MVC como patrón de arquitectura para implementar sistemas web. En Rails los modelos se definen haciendo uso de la biblioteca *ActiveRecord*, la cual implementa el patrón de mapeo objeto-relacional (*Object-relational mapping*, ORM) esto facilita el acceso de información contenida en bases de datos relacionales, dado que es el caso típico en sistemas web convencionales.

En Rails, la vista es responsable de la creación de la respuesta dada para ser mostrada en un navegador. En su forma mas simple, una vista «es un trozo de código HTML que muestra un texto fijo» [22]. Frecuentemente se requiere mostrar contenido dinámico creado por una acción en un controlador, por lo que el contenido dinámico es generado por medio de plantillas, el esquema más común de plantillas es llamado Ruby Embebido (*Embedded Ruby*, ERB), el cual inserta pedazos de código de Ruby dentro de una vista, similar a la forma como se hace en otros marcos de trabajo como PHP o JSP. También se pude hacer uso de Ruby Embebido para incrustar pedazos de código Javascript en el servidor, que serán ejecutados en el navegador, lo cual permite crear interfases dinámicas haciendo uso de *Asynchronous Javascript and XML* (AJAX).

Finalmente en Rails los controladores son el centro lógico del sistema, ya que coordinan la interacción entre el usuario, las vistas y el modelo [22].

2.1.2. MVC y PEAT

`PEAT` saca provecho del patrón MVC de las siguientes maneras [22]:

1. Modelo: dado que en un principio los servicios web de recomendaciones y desagregación estaban en construcción se utilizó la biblioteca *ActiveRecord* con el fin de tener datos reales estáticos que permitieran la implementación de la interfaz de usuario. Posteriormente se reemplazaron éstos por nuevos modelos que hacían uso de los servicios web del *backend*, por medio de la biblioteca `Bezel`. Estos cambios no tuvieron un alto impacto en el desarrollo del sistema per se, dado el desacoplamiento existente entre los modelos y las vistas.

2. Vista: haciendo uso de plantillas se generan representaciones de los principales modelos del sistema en HTML y JSON.

   Sin embargo, para ciertos modelos, como las *recomendaciones* y los *reportes de consumo*, se contaba con una tercera representación en forma de PDF del modelo.

2.2. Servicios web RESTful

La *World Wide Web Consortium*, W3C, define que un servicio web en general es un sistema de software diseñado para dar soporte a interacciones máquina-máquina a través de una red informática [24]. Su implementación nació de tener diferentes sistemas que puedan intercambiar datos entre ellos.

Transferencia de Estado Representacional (*Representational State Transfer*, REST) es una arquitectura de software para la implementación de servicios web. En REST se define la existencia de recursos (elementos de información), donde cada recurso tiene un conjunto de representaciones posibles. Por ejemplo una lista de errores por arreglar (recurso) puede ser presentado en forma de un documento XML, una página HTML o un archivo CSV (representaciones).

Se tienen cuatro características principales [15]:

- Protocolo cliente/servidor sin estado (*stateless*): cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Esto implica que ni el cliente ni el servidor necesitan recordar ningún tipo de estado.
- Conectividad (*connectedness*): las representaciones son un hipermedio en el cual se tienen ligas a otros recursos, como resultado de esto, es posible navegar de un recurso REST a muchos otros, sin necesidad de una infraestructura adicional.
- Direccionabilidad (*addressability*): la capacidad para identificar los recursos del sistema. Cada recurso es direccionable únicamente a través de su Identificador de Recursos Uniforme (*Uniform Resource Identifier*, URI).
- Interfaz uniforme (*uniform interface*): se tiene un conjunto de operaciones bien definidas que se aplican a todos los recursos del sistema. Se usan los métodos de HTTP para definir las operaciones más importantes como son GET, POST, PUT, PATCH y DELETE.

Los servicios web que implementan una arquitectura REST se suelen llamar servicios web RESTful.

2.2.1. REST y Rails

La arquitectura REST es parte vital de Rails, todo el enrutamiento y manejo de peticiones se basa en esta arquitectura.

En REST se hace uso de un conjunto finito de verbos para operar sobre otro conjunto de objetos. Dado que estamos usando HTTP como capa de transporte, los verbos

corresponden a los métodos HTTP (GET, POST, PUT, PATCH, y DELETE). Los objetos corresponden a los recursos del sistema, los cuales son etiquetados usando un (*Uniform Resource Locator*, URL).

Un navegador solicita páginas de Rails al hacer una petición para una dirección URI haciendo uso de un método HTTP específico, como GET o POST. Cada método es una petición para realizar una operación sobre el recurso.

Haciendo uso de la interfaz uniforme, Rails define todo un conjunto de rutas para un recurso, tomando como ejemplo el concepto de edificio (*Building*), entonces se define el recurso y sus rutas asociadas con lo siguiente:

```
resources :buildings
```

En la Tabla 2.1 se pueden ver las rutas y métodos asociados para las principales operaciones sobre el recurso *buildings*, esto acelera en gran medida el desarrollo de servicios web.

Método HTTP	Ruta	Controlador#Acción	Usado para
GET	/buildings	buildings#index	Listar todos los edificios
GET	/buildings/new	buildings#new	Regresar una forma para crear un edificio
POST	/buildings	buildings#create	Crear un nuevo edificio
GET	/buildings/:id	buildings#show	Mostrar un edificio especifico
GET	/buildings/:id/edit	buildings#edit	Regresar una forma para editar un edificio
PATCH	/buildings/:id	buildings#update	Actualizar un edificio
DELETE	/buildings/:id	buildings#destroy	Borrar un edificio

Tabla 2.1: Rutas y métodos para el recurso *buildings* [6].

2.2.2. REST y PEAT

La arquitectura REST influye en las tres partes principales de PEAT.

- *Frontend*: por medio de Rails se implementa un servicio web RESTful para proveer de información a la interfaz gráfica del sistema.
- *Backend*: los servicios web que componen el *backend*, como el sistema de recomendaciones, son del tipo RESTful. El recurso principal del sistema es el de edificio (*Building*) el cual se encuentra alrededor de los demás recursos del sistema.

- *Middleware*: la biblioteca `Bezel` se beneficia gracias al hecho de que el *backend* sea de tipo RESTful, pues facilita la integración del sistema, dado que hace un mapeo casi directo de los recursos a clases y objetos.

2.3. Desarrollo guiado por pruebas y comportamiento

El tener código fuente que sea limpio y que además tenga la funcionalidad deseada es uno de los principales objetivos en cualquier sistema. Cabe señalar que por código limpio se entenderá en este trabajo, aquel código fuente que sea fácil de entender y de modificar [2].

El desarrollo guiado por pruebas (*Test Driven Development*, TDD) y el desarrollo guiado por comportamiento (*Behavior Driven Development*, BDD) son prácticas de ingeniería de software que tiene por objetivo el obtener código limpio y funcional.

2.3.1. Desarrollo guiado por pruebas (TDD)

El desarrollo guiado por pruebas (*Test Driven Development*, TDD) es una buena práctica de software que involucra el escribir las pruebas antes de realizar la implementación, esto mejora el diseño y eficacia de la implementación. Al escribir las pruebas antes que el código fuente, contrario al proceso habitual, permite que las pruebas ayuden a guiar el diseño del código fuente en pequeños pasos. En el largo plazo este proceso implementa un código fuente bien estructurado que es fácil de mantener y de modificar[2].

El proceso clásico en TDD es el siguiente [21]:

1. Implementar una prueba: esta prueba debe ser breve y debe probar solamente una unidad de código (función, clase o módulo).

2. Asegurar que la prueba falle: se verifica que la prueba falle antes de escribir cualquier código. Esto es para asegurar que la prueba realmente hace lo que se espera de ella.

3. Implementar la nueva funcionalidad: se implementa el mínimo código de forma que la prueba pase satisfactoriamente.

4. Mejorar el código fuente: se elimina toda repetición que se originó al implementar la nueva funcionalidad para lograr que la prueba pase satisfactoriamente, dentro de esta fase también se realiza cualquier otra optimización y/o abstracción necesaria. Este proceso recibe el nombre de refactorización, el cual es parte vital del proceso.

Este proceso se repite hasta que se termina de implementar toda la funcionalidad del sistema. Al seguir este proceso, en teoría, se asegura que el código fuente siempre se mantiene lo mas simple posible y se encuentra verificado completamente.

Ventajas

TDD va mas allá de la mera verificación al hacer uso de las pruebas para mejorar la estructura del código fuente, por esto es que TDD es una práctica de software y no solamente una herramienta de verificación.

Al alinear continuamente el código fuente con las pruebas se obtiene código fuente conformado por pequeños métodos, cada uno de los cuales tiene solamente una sola responsabilidad. Estos métodos tienden a tener bajo acoplamiento y con pocos efectos laterales, lo que facilita su comprensión y mantenimiento.

Desventajas

TDD no es un substituto para pruebas de aceptación, es decir, pruebas que confirman que el software funciona según los requerimientos del cliente.

TDD asume que se conoce el resultado esperado que se quiere verificar, cuando los requerimientos no son totalmente claros, entonces TDD no resulta adecuado puesto que es difícil escribir las pruebas de un proceso que no se conoce en detalle.

RSpec

RSpec es un marco de trabajo del proceso TDD/BDD para el lenguaje de programación Ruby. *RSpec* define un lenguaje de dominio específico (DSL) para implementar pruebas unitarias y de aceptación [3].

Un ejemplo de una prueba unitaria haciendo uso de *RSpec*:

```
1  describe Bezel::Client do
2    # Contexto de la prueba, teniendo el API versión 2 del
       servidor C3.
3    context "v2" do
4
5      # Nombre de la prueba
6      it "escapes unicode sequences" do
7        body = Bezel.client.send(:generate_body,{"value"=>"Dé
       connexion"})
8
9        # Aserciones sobre el comportamiento esperado.
10       body.should_not include('é')
11       body.should include('\u00e9')
12     end
13   end
14 end
```

El método *it* (en la línea 6) crea un ejemplo del comportamiento de *Bezel::Client* dentro de un contexto, el cual es que el servidor C3 esta corriendo la versión 2.

2.3.2. Desarrollo guiado por comportamiento (BDD)

En desarrollo guiado por comportamiento (*Behavior Driven Development*, BDD) es una práctica de software surgida de el TDD, al igual que esta última involucra escribir

las pruebas antes de escribir el código fuente y la refactorización contínua del código fuente. Así mientras que el TDD se enfoca en describir el comportamiento de unidades de código el BDD se enfoca en describir el comportamiento de la interacción entre varios módulos del sistema.

En BDD la unidad de prueba, es conocida como las pruebas de aceptación, las cuales definen una serie de escenarios, los cuales surgen de los casos de uso del sistema. Estos escenarios casi siempre se especifican en un lenguaje DSL que hace uso de lenguaje natural (casi siempre inglés) para definir así el comportamiento esperado del sistema [3].

Ventajas

En BDD el diseño del código fuente del sistema se guía a través de las pruebas de aceptación desarrolladas junto con el cliente, por lo que obtenemos software que funciona según los requerimientos del cliente.

Desventajas

BDD no es un substituto para pruebas unitarias, dado que el enfoque de las pruebas de aceptación es sobre el comportamiento de grandes bloques del sistema y no en los detalles del mismo.

Para obtener el mayor beneficio con BDD es necesario especificar las pruebas de aceptación con el cliente y gente relacionada con el negocio, lo cual no siempre es posible [21].

Cucumber

Cucumber es un marco de trabajo que permite especificar y ejecutar pruebas de aceptación siguiendo el proceso BDD. Define un lenguaje llamado *Gherkin* que permite escribir escenarios en lenguaje natural. De esta manera facilita que personas no técnicas, como el cliente o gente de negocio, puedan ayudar a escribir o dar el visto bueno a las pruebas de aceptación [3].

Un ejemplo de prueba de aceptación es el siguiente:

```
1  Característica:
2    Para disminuir mis costos de energía eléctrica
3    Como un usuario del sistema PEAT
4    Quiero saber sobre las posibles mejoras a mi edificio
5
6    Escenario: Desplegar la lista de recomendaciones
7      Dado que ingresó al sistema como el usuario @juan por
       primera vez
8      Y que tengo una cuenta y un edificio
9      Cuando entró a la página de recomendaciones
10     Entonces Yo debo ver un botón de "Añadir al plan" en cada
       recomendación
```

Gherkin es un lenguaje que usa la identación para definir su estructura, de manera que los saltos de línea dividen las diferentes declaraciones, la mayoría de las líneas empiezan con palabras clave.

En los escenarios se tiene tres principales palabras clave:

- Dado (*Given*): el propósito de esta declaración es el poner el sistema en el estado deseado para iniciar la prueba correspondiente, antes de que el usuario (o un sistema externo) interactué con el sistema.

- Cuando (*When*): el propósito de esta declaración es el describir la acción que se realiza por el usuario, la cual vamos a probar.

- Entonces (*Then*): el propósito de esta declaración es observar los resultado de la acción realizada, estas observaciones debe ser visibles para el usuario o sistema externo.

2.3.3. TDD/BDD y PEAT

Dado que TDD y BDD se enfocan en el comportamiento a diferentes niveles de un sistema éstos son complementarios y permiten subsanar sus respectivas desventajas. Así que en la implementación del sistema PEAT está basado en TDD y BDD para guiar el diseño del sistema, haciendo uso de los marcos de trabajo *RSpec* y *Cucumber* para realizar la especificación de las pruebas unitarias y de aceptación del sistema.

Cabe mencionar que el uso de pruebas de aceptación permite a su vez que la transición del uso de datos estáticos al uso del API final sea lo mas fluído posible, ya que se indicaban los puntos críticos en los que se tiene que realizar cambios para obtener nuevamente el comportamiento esperado por el cliente.

2.4. Ruby, metaprogramación y lenguajes DSL

El lenguaje de programación Ruby es un lenguaje dinámico con una gramática compleja pero expresiva y cuenta con una biblioteca estándar extensa y poderosa. Ruby toma inspiración de otros lenguajes de programación como Lisp, Smalltalk y Perl [8].

Ruby es un lenguaje orientado a objetos puro, sin embargo permite el uso de otros paradigmas como el funcional o imperativo, incluye capacidades para la metaprogramación las cuales son usadas para crear fácilmente lenguajes de dominio especifico (*Domain Specific Language*, DSL)[8].

2.4.1. Metaprogramación y Ruby

La metaprogramación consiste en escribir programas que implementan o manipulan otros programas (o asimismos). Los ejemplos más comúnes de metaprogramación son los compiladores y generadores de código automatizado.

En Ruby la metaprogramación se enfoca en código que se manipula asimismo en tiempo de ejecución, a este tipo de metaprogramación se le da el nombre de metaprogramación dinámica, para así diferenciarla de la metaprogramación asociada típicamente a los generadores de código y compiladores [18].

Ruby es un lenguaje de programación muy dinámico:

- Permite crear clases y módulos en tiempo de ejecución.
- Permite agregar nuevos métodos a clases en tiempo de ejecución.
- Define varias llamadas (*callbacks*) a eventos relacionados con clases, módulos y métodos.

También se tiene un API de reflexión, el cual permite que un programa examine su estado y estructura, además permite que un programa altere su propio estado y estructura.

El API de reflexión junto con el uso de bloques e iteradores y el uso opcional de paréntesis para la aplicación de funciones hacen de Ruby un lenguaje ideal para la metaprogramación [8].

2.4.2. Lenguajes de dominio especifico

Lenguajes de programación como Ruby o C++ son Lenguajes de Propósito General (*General Purpose Language*, GPL), es decir, lenguajes que son diseñados para ser usados para implementar software en una gran variedad de dominios.

En contrapuesto se tienen los Lenguajes de Dominio Específico (*Domain Specific Language*, DSL) los cuales están diseñados para resolver problemas en un dominio específico [14].

DSL internos y externos

Para implementar un DSL se tienen dos rutas:

- Definir la gramática del lenguaje haciendo uso de herramientas como ANTLR o Yacc, los cuales son DSL para escribir analizadores sintácticos de lenguajes. Posteriormente se tiene que escribir un intérprete para la gramática definida. Este tipo de lenguajes se les conoce como DSL externos.
- Tomar un lenguaje GPL y modificarlo para que se asemeje al DSL que se necesita. A este tipo de DSL se les conoce por el nombre de DSL internos.

En el lenguaje de programación Ruby se tiene una gran cantidad de lenguajes DSL internos dada la flexibilidad de la sintaxis del lenguaje.

En el siguiente ejemplo se tiene código Ruby que usa la biblioteca *Markaby* para generar HTML. En este caso se tiene un DSL para generar HTML haciendo uso de Ruby como lenguaje GPL base.

```
require 'markaby'

html = Markaby::Builder.new do
  head {  title "Mi página personal"}
  body do
    h1 "Bienvenido a mi página personal"
    b "Mis pasatiempos"
    ul do
      li "Correr"
      li "Leer"
      li "Programación"
    end
  end
end
```

El ejemplo anterior muestra un código válido en Ruby, pero tiene la forma de un lenguaje específico para generar HTML. *Markaby* define un DSL interno, ya que hace uso de Ruby para implementar su DSL.

Una gran ventaja de los DSL internos es que se puede hacer uso del GPL subyacente fácilmente cuando sea necesario, aunque la sintaxis del DSL interno esta limitado por la sintaxis del lenguaje GPL. Este puede ser un gran limitante en otros lenguajes como en Java, en el cual es posible implementar DSL internos [14]. Cabe mencionar que éstos no dejan de tener un gran parecido a Java por lo rígido de su sintaxis, mientras que la sintaxis de Ruby es flexible lo que permite que los DSL internos definidos con este se vean mas adaptados al problema en cuestión.

En este capítulo se vieron tanto patrones de arquitectura como prácticas de software que influyeron en gran medida en la implementación del sistema `PEAT`. También se expuso el concepto de metaprogramación y sus beneficios en contexto del lenguaje de programación Ruby. La aplicación de la metaprogramación fue pieza clave para la rápida implementación de la biblioteca `Bezel` como se mostrará en la Sección 3.4.

2.5. Conocimiento obtenido durante la carrera

Durante la implementación del sistema `PEAT` hice uso del conocimiento obtenido durante mi paso por la carrera de Ciencias de la Computación, materias como Matemáticas Discretas y Análisis Lógico me han servido para ser un mejor programador. El resto de las materias relacionadas con matemáticas puras me ayudaron a tener en mente varias forma de abordar y resolver los problemas que se presentaban durante la implementación del sistema PEAT.

De las materias restantes adquirí conceptos y conocimiento que fueron de gran utilidad para la implementación de `PEAT`:

- Lenguajes de Programación: los conceptos adquiridos en esta materia me han permitido aprender rápidamente nuevos lenguajes de programación, también el saber los diferentes paradigmas (estructurado, funcional, orientado a objetos) permite obtener la mayor productividad en relación al problema a solucionar. El concepto de lenguaje de dominio especifico (LDE) fue de vital importancia para

la implementación exitosa del sistema PEAT puesto que permitió el diseño e implementación de la biblioteca Bezel la cual define un LDE para definir modelos y sus asociaciones en base a tipos obtenidos del servidor C3.

- Sistema Operativos: el conocimiento obtenido sobre manejo de procesos y memoria fue pieza fundamental para obtener mejoras en el tiempo de inicialización del sistema PEAT y de su rendimiento en general.
- Computación Distribuida: me dio herramientas para poder solucionar problemas que se presentan en sistemas distribuidos como sincronización de sesiones y datos, permitiendo conocer las ventajas y desventajas de hacer uso de memoria compartida distribuida para comunicación de sesiones y datos entre las diferentes instancias del sistema PEAT.
- Redes de Computadoras: al tener un conocimiento de los protocolos TCP/IP y del protocolo HTTP permite el uso óptimo de la arquitectura REST.
- Ingeniería de software: permitió ver la necesidad de detalle en varios de los requerimientos iniciales, detectar deficiencias en los primeros borradores de los casos de uso del sistema.

Capítulo 3

Diseño e implementación del sistema PEAT

Para lograr los objetivos del sistema PEAT se dividió en cuatro componentes principales:

- Perfil sencillo del edificio (*Simple Building Profile*): este componente realiza la configuración inicial del edificio, realizando solamente un número limitado de preguntas básicas para definir las características básicas del mismo.
- Perfil detallado del edificio (*Detailed Building Profile*): este componente se encarga del manejo de las características básicas, detalladas y opcionales de un edificio.
- Recomendaciones (*Recommendations*): este componente genera una lista de medidas de ahorro de energía considerando la información ingresada al sistema hasta el momento. Además debe permitir al usuario obtener información detallada sobre las recomendaciones dadas y permitir que el usuario se comprometa a llevar acabo una recomendación.
- Plan de ahorro de energía (*Energy Savings Plan*): este componente presenta el consumo de energía del edificio al usuario, también presenta las recomendaciones que el usuario se ha comprometido a poner en acción y el ahorro que representa llevar acabo estas medidas.

El modelo de edificio es el principal modelo dentro de PEAT, por lo que obtener una correspondencia correcta entre un edificio y su consumo energético es de vital importancia para dar la información y recomendaciones más útiles y fidedignas al usuario.

3.1. Modelo de información

El objetivo principal del modelo de información de PEAT es definir el modelo de edificio y su relación con el consumo energético del usuario. Así tenemos el concepto

de edificio el cual es una estructura independiente a la que se le esta proporcionando un servicio ya sea de electricidad o gas.

3.1.1. Estructura de facturación de PG&E

Para obtener esta correspondencia entre edificio-consumo es necesario tener en cuenta la estructura de facturación usada por PG&E puesto que de esta estructura es que se obtiene el consumo de electricidad y gas.

Los modelos se pueden categorizar en dos grupos: demográficos o geográficos.

El usuario, la cuenta y el contrato de servicio representan el lado demográfico del modelo de información:

- Usuario: representa al empresario, el cual puede tener una o más cuentas.
- Cuenta: representa la contabilidad financiera del usuario. Tiene uno o más contratos de servicio, uno por tipo de servicio (electricidad o gas), también se tiene una o más direcciones de servicio asociadas.
- Contrato de servicio: representa los servicios que el cliente adquiere de PG&E, como la electricidad o el gas y puede tener uno o más puntos de servicio asociados.

La dirección de servicio, el punto de servicio y el medidor representan la parte geográfica del modelo de información:

- Dirección de servicio: es una ubicación física en donde se prestan los servicios, en PEAT son el modelo que se quiere asociar al modelo de edificio para obtener el consumo del edificio, el cual puede tener uno o más puntos de servicio.
- Punto de servicio: es una coordenada geográfica en donde se conectan los servicios y puede tener uno o más medidores asociados.
- Medidor: es un dispositivo instalado en un punto de servicio que registra el consumo del servicio proporcionado.

La relación entre todos estos modelos se puede ver en la Figura 3.1.

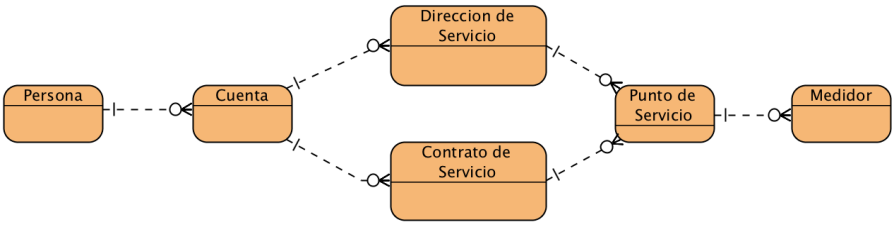

Figura 3.1: Modelos principales para la facturación en PG&E.

3.1.2. Relación Edificio-Facturación

Para que PEAT pueda cumplir sus requerimientos funcionales es necesario definir claramente la relación entre un edificio y su consumo energético calculado por uno o varios medidores. A pesar de que los medidores mantienen la información de consumo, el relacionar directamente un medidor con un edificio no es una tarea sencilla para un usuario, porque aunque el usuario generalmente conoce el número de medidores que tiene no conoce los identificadores para cada uno de éstos.

Dado el escenario anterior para facilitar la creación del perfil de un edificio se solicita al usuario que haga la correspondencia entre las direcciones de servicio asociadas a su cuenta y el perfil de edificio que se está creando. El usuario al establecer esta correspondencia permite al sistema obtener de forma automatizada los puntos de servicio y sus medidores asociados al edificio, con el objetivo de obtener su consumo energético real.

3.2. Casos de uso

3.2.1. Diagrama general

En la Figura 3.2 se muestra el diagrama general de casos de uso del sistema PEAT. Se tienen diez casos de uso en total, siendo siete de éstas definidas para PEAT y tres casos para la biblioteca Bezel.

Los casos de uso se agrupan en los cuatro componentes principales de la siguiente manera:

- Perfil sencillo del edificio (*Simple Building Profile*): Tiene solo el caso de uso *Crear perfil del edificio* pero este caso de uso es uno de los más complejos en cuestión de interacción con el usuario y su importancia para obtener las características básicas del edificio.
- Perfil detallado del edificio (*Detailed Building Profile*): Contiene los casos de uso *Administrar perfil del edificio* y *Obtener información del edificio* los cuales se encargan de la administración y obtención de información sobre el edificio.
- Recomendaciones (*Recommendations*): Contiene los casos de uso *Administrar recomendaciones* y *Ver recomendación*.
- Plan de ahorro de energía (*Energy Savings Plan*): Contiene los casos de uso *Ver historial de consumo* y *Administrar plan de ahorro*, que se enfocan a la visualización del consumo eléctrico del usuario y las posibles recomendaciones para disminuir su consumo.

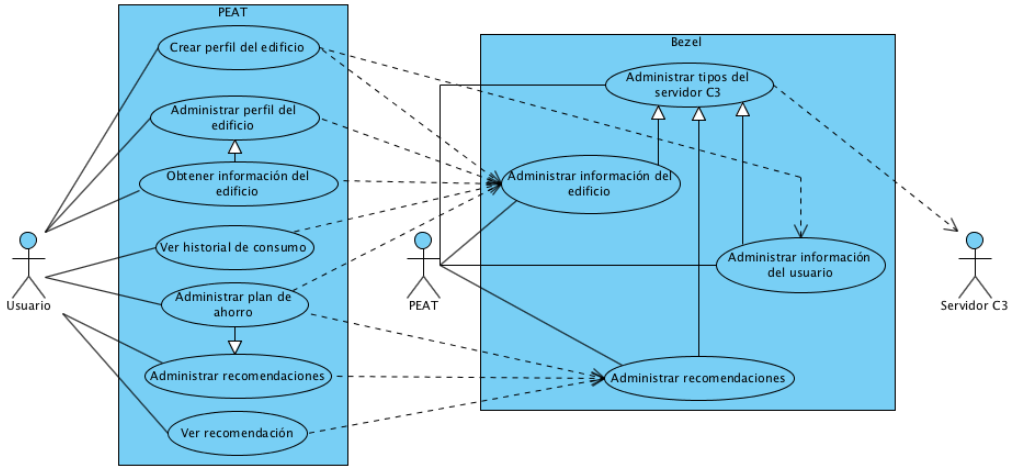

Figura 3.2: Diagrama general de casos de uso de PEAT.

3.2.2. Crear perfil del edificio

Para crear el perfil inicial de un edificio es necesario obtener las siguientes características:

- La industria que mejor describe el negocio del usuario.
- El tamaño del edificio (área).
- El tipo de edificio.
- La(s) dirección(es) de servicio asociada(s) al edificio.

Todas estas características se consideran básicas, es decir, vitales para el funcionamiento del sistema por lo que todas son requeridas para construir el perfil del edificio.

En este caso de uso también se obtiene la relación entre el edificio y su consumo a partir de las direcciones de servicio que el usuario asocie al perfil.

El caso de uso esta diseñado de forma que el proceso sea realizado solamente una vez por edificio. Las características básicas se mantienen accesibles por medio del perfil detallado del edificio.

Caso de Uso:	**Crear perfil del edificio.**
Descripción:	Un usuario PyME ingresa al sistema para crear un perfil inicial de un edificio, respondiendo ciertas preguntas básicas sobre el edificio.

Actor:	Usuario PyME
Precondiciones:	▪ El usuario ha sido autentificado por medio de un elemento (*token*) de autenticación (*Single Sign On*, SSO) asignado por el sistema de PG&E. ▪ Los datos de intervalo y de facturación del usuario se encuentran en el sistema.
Requerimientos no funcionales:	▪ El usuario debe ser capaz de completar el perfil del edificio en 20 segundos o menos.

Flujo normal:

1. El sistema checa si el usuario tiene más de una cuenta.
   a) Si hay más de una cuenta el sistema despliega una lista de cuentas, y el sistema requiere que el usuario elija una cuenta.
2. El sistema despliega información sobre la cuenta: número de cuenta y direcciones de servicio asociadas.
   a) Si la cuenta tiene más de una dirección de servicio el sistema requiere que el usuario seleccione por lo menos una dirección de servicio.
3. El usuario selecciona las direcciones de servicio que apliquen al edificio.
4. El usuario puede proporcionar un apodo para el edificio.
5. El usuario debe indicar la industria que mejor describe su negocio.
6. El usuario debe ingresar la siguiente información sobre el edificio:
   a) Tipo.
   b) Rango de tamaño.
   c) Antigüedad.
7. El usuario puede enviar los datos ingresados o cancelar.
8. El usuario puede añadir otro edificio o si ya tiene al menos un perfil de edificio asociado puede proceder al plan de ahorro.

Postcondiciones:	Se genera un identificador único para el perfil del edificio generado, el cual está asociado a los datos de facturación de la cuenta seleccionada por el usuario PyME.

3.2.3. Administrar perfil del edificio

Para el sistema PEAT se tiene las siguientes características:

- Básicas: se refiere a las características esenciales de un edificio: tipo, tamaño y antigüedad, las cuales son vitales para el funcionamiento del sistema.
- Detalladas: son las características más significativas y fáciles de responder. Por ejemplo: horas de operación, número de empleados, número de pisos, etcétera.
- Opcionales: son características técnicas más avanzadas sobre el equipo y estructura de un edificio.

Las características básicas son definidas al crear el perfil inicial de un edificio como se documenta en el caso de uso anterior. Las características detalladas y opcionales son definidas por medio de dos rutas:

- Por medio del perfil detallado del edificio, en la que se despliega las preguntas y respuestas sobre las características del edificio. Esta ruta esta documentada en este caso de uso.
- Un componente que es embebido en varias partes del sistema que realiza una sola pregunta al usuario para ir obteniendo progresivamente más información sobre el edificio. Este componente llamado componente de ingreso progresivo (*Progressive Profile Widget* PPW). Esta ruta es documentada en el caso de uso *Obtener información del edificio*.

Caso de Uso:	**Administrar perfil del edificio.**
Descripción:	El usuario tiene acceso al perfil completo del edificio, es decir a todas las preguntas y respuestas asociadas a las características básicas, detalladas y opcionales de un edificio.
Actor:	Usuario PyME

Precondiciones:	▪ El usuario ha sido autentificado por medio de un elemento (*token*) de autenticación (*Single Sign On* SSO) asignado por el sistema de PG&E. ▪ El usuario debió haber creado previamente el perfil básico del edificio.
Requerimientos no funcionales:	▪ Todas las acciones realizadas por el usuario deben ser procesadas en menos de un segundo.

Flujo normal:

1. El sistema verifica si la cuenta tiene más de un edificio asociado.
   - Si hay más de un edificio asociado a la cuenta, el usuario tiene que seleccionar el edificio correspondiente.
2. El sistema despliega una lista de preguntas y respuestas, dividida en secciones. Las secciones completadas están marcadas con una marca de terminado.
   - Las secciones que se despliegan dependen del tipo de edificio. Por ejemplo: construcción, iluminación, calefacción y refrigeración.
   - El usuario puede ver y actualizar sus respuestas a todas las preguntas en una sola pantalla.
3. El usuario selecciona una sección, esta se expande para mostrar todas las preguntas de la misma.
4. El usuario responde una o varias preguntas.
   a) Las preguntas son ordenadas dentro de cada sección según un orden predefinido.
   b) Según vaya respondiendo el usuario se va actualizando un indicador del progreso de llenado del perfil. Cada pregunta con su respuesta tiene el mismo valor para calcular el progreso de llenado.
5. El usuario puede elegir continuar con la siguiente sección o seleccionar en el encabezado otra sección.
6. El sistema almacena la información en cuanto el usuario selecciona una respuesta. Al obtener nueva información el sistema debe recalcular las recomendaciones asociadas al edificio.

Postcondiciones:	▪ Un perfil del edificio más detallado si el usuario dio respuesta a una pregunta sin contestar sobre su edificio. ▪ Si el usuario dio nueva información se recalculan las recomendaciones asociadas al edificio.

3.2.4. Obtener información del edificio

Uno de los requerimientos más importante para PEAT es el incentivar y obtener más información sobre las características de los edificios del usuario, ya que entre más datos se obtenga de éstos se podrán generar mejores planes de ahorro y recomendaciones para el usuario.

La forma en como se aborda esta necesidad es por medio de la implementación de un componente que esta presente en las dos partes principales del sistema: el plan de ahorro y la lista de recomendaciones. Este elemento llamado componente de ingreso progresivo (*Progressive Profile Widget*, PPW), tal componente realiza una sola pregunta al usuario obteniendo progresivamente mas información sobre el edificio.

Caso de Uso:	**Obtener información del edificio.**
Descripción:	El sistema obtiene información progresivamente del usuario haciendo uso de un componente que realiza una pregunta al usuario en varias partes del sistema. Este es conocido como componente de ingreso progresivo (*Progressive Profile Widget*, PPW).
Actor:	Usuario PyME
Precondiciones:	▪ El usuario ha sido autentificado por medio de un elemento (*token*) de autenticación (*Single Sign On* SSO) asignado por el sistema de PG&E. ▪ El usuario debió haber creado previamente el perfil básico del edificio.
Requerimientos no funcionales:	Las actualizaciones a la página deben hacerse en forma asíncrona y sin recargar la página completa.

Flujo normal:

1. El sistema despliega el componente PPW en la parte izquierda en las pantallas de plan de ahorro y de recomendaciones.
2. Dentro del PPW, se presenta al usuario una pregunta a responder:
   a) Dependiendo de la pregunta, el usuario puede seleccionar una respuesta de una lista desplegable, seleccionar casillas, responder si o no con un botón radial o ingresar un valor en un campo.
   b) Cada tipo de edificio tiene una lista predefinida ordenada de preguntas a ser desplegadas en el componente.
   c) Algunas preguntas tienen imágenes asociadas para facilitar al usuario su respuesta.
3. El usuario puede responder la pregunta o saltar la pregunta o no hacer nada en el PPW.
   a) Si el usuario responde:
      1) La respuesta provoca un refinamiento del perfil del edificio y el nuevo cálculo de las recomendaciones propuestas.
         I. El perfil del edificio es actualizado, las recomendaciones son recalculadas.
         II. El sistema despliega un indicador de progreso mientras se recalculan las recomendaciones.
         III. Al finalizar el refinamiento el orden de las preguntas o los valores del plan de ahorro deberán ser actualizados, sin recargar la página completa.
      2) El sistema despliega la siguiente pregunta.
      3) El sistema almacena la respuesta dada.
   b) Si el usuario elige saltar la pregunta entonces el sistema despliega la siguiente pregunta sin necesitar de una respuesta por el usuario.
4. Desde el PPW, el usuario puede seleccionar «Building Profile» para ver el perfil detallado del edificio.

Postcondiciones:

- Un perfil del edificio más detallado si el usuario dio respuesta a la pregunta desplegada sobre su edificio.
- Si el usuario dio nueva información se recalculan las recomendaciones asociadas al edificio.

3.2.5. Administrar recomendaciones

En los casos de uso anteriores se obtiene del usuario el perfil del edificio, al menos sus características básicas. Haciendo uso de esta información el sistema PEAT genera recomendaciones sobre como disminuir el consumo de energía a través de cambios de comportamiento y/o modernización de equipo.

Las recomendaciones se muestran en tres partes del sistema:

1. En el plan de ahorro.
2. En las recomendaciones.
3. En el perfil detallado del edificio.

El comportamiento general al administrar las recomendaciones en estas tres secciones es definido en este caso de uso, teniéndose en los casos de uso posteriores los detalles particulares del comportamiento para esa sección.

Caso de Uso:	**Administrar recomendaciones.**
Descripción:	El sistema despliega una lista de todas las recomendaciones que se ajustan al perfil del edificio.
Actor:	Usuario PyME
Precondiciones:	▪ El usuario ha sido autentificado por medio de un elemento (*token*) de autenticación (*Single Sign On* SSO) asignado por el sistema de PG&E. ▪ El usuario debió haber creado previamente el perfil básico del edificio.
Requerimientos no funcionales:	▪ Cualquier acción en una recomendación debe hacerse de forma asíncrona y sin necesitar de recargar la página completa.

Flujo normal:

1. El sistema despliega las recomendaciones, en el plan de ahorro, la lista de recomendaciones o en el perfil detallado del edificio.
2. El usuario puede tomar las siguientes acciones:
   - Añadir al plan (de ahorro).
     a) Cuando el usuario selecciona «Añadir al plan», una ventana modal se abre y pregunta: «¿ Cuando piensa completar esta acción ?».
     b) El sistema proporciona lista predefinida de rangos de tiempo: una semana, un mes, tres meses, etcétera.
     c) El usuario elige el rango de tiempo adecuado y selecciona la opción «Añadir al plan».
     d) El sistema añade la recomendación al plan de ahorro del edificio.
   - No aplica.
     a) El sistema mueve la recomendación al final de la lista de recomendaciones.
     b) En cualquier momento, el usuario puede cambiar el estado de la recomendación, es decir, añadir la recomendación al plan de ahorro o indicar que no es aplicable al edificio.
   - Ya terminado.
     a) El sistema mueve la recomendación al final de la lista de recomendaciones.
     b) En cualquier momento, el usuario puede cambiar el estado de la recomendación, es decir, añadir la recomendación al plan de ahorro o indicar que no es aplicable al edificio.
3. El sistema refina y reordena la lista de recomendaciones después de que el usuario realiza cualquiera de las acciones anteriores.

Flujo alternativo:

1. El usuario contesta alguna pregunta en el PPW o en el perfil detallado del edificio.
2. El sistema refina y reordena la lista de recomendaciones después de que el usuario realiza cualquiera de las acciones anteriores.

Postcondiciones:	▪ Si el usuario añadió una recomendación a su plan de ahorro, entonces se tiene un plan de ahorro actualizado. ▪ Si el usuario realizó cualquier acción en una recomendación, entonces la lista de recomendaciones es actualizada y reordenada.

3.2.6. Ver recomendación

En el plan de ahorro, la lista de recomendaciones o en el perfil detallado del edificio solo se despliega el título y una breve descripción de cada recomendación. Para dar mayor información sobre la recomendación a implementar se proporciona al usuario más información sobre los pasos para lograr su implementación.

Caso de Uso:	**Ver recomendación**
Descripción:	El sistema despliega información adicional sobre una recomendación para que el usuario pueda elegir implementarla.
Actor:	Usuario PyME
Precondiciones:	▪ El usuario ha sido autentificado por medio de un elemento (*token*) de autenticación (*Single Sign On* SSO) asignado por el sistema de PG&E. ▪ El usuario debió haber creado previamente el perfil básico del edificio.
Requerimientos no funcionales:	Cualquier acción en una recomendación debe hacerse en forma asíncrona y sin necesitar una recarga completa de la página.

Flujo normal:

1. El usuario ve una recomendación que capta su atención y selecciona el título de la recomendación.
2. El sistema despliega información adicional sobre la recomendación. Por ejemplo, (1) cuanto ahorraría al realizar tal recomendación, (2) qué se necesita hacer para obtener este ahorro, etcétera.
3. El usuario puede realizar las acciones definidas en el caso de uso anterior, es decir, añadir la recomendación al plan de ahorro, indicar que no aplica dicha recomendación o bien la recomendación ha sido completada.

Postcondiciones:	Si el usuario realizó cualquier acción en una recomendación, entonces la lista de recomendaciones es actualizada y reordenada.

3.2.7. Ver historial de consumo

Caso de Uso:	**Ver historial de consumo**
Descripción:	Se muestra el consumo energético del edificio mediante un conjunto de gráficas con múltiples vistas para diferentes rangos de tiempo y categorías de gasto (gas, electricidad y emisiones de carbón).
Actor:	Usuario PyME
Precondiciones:	▪ El usuario ha sido autentificado por medio de un elemento (*token*) de autenticación (*Single Sign On* SSO) asignado por el sistema de PG&E. ▪ El usuario debió haber creado previamente el perfil básico del edificio.
Requerimientos no funcionales:	Las actualizaciones a la página deben hacerse en forma asíncrona y sin necesitar una recarga completa de la página.

Flujo normal:

1. El sistema despliega inicialmente una gráfica del gasto anual de electricidad y gas del edificio.
2. El sistema muestra un menú con las siguientes opciones de visualización:
   - Gasto: el cual es calculado en dolares con gráficas anuales, mensuales y diarias.
   - Electricidad: se refiere al cálculo en kilowatt-hora con gráficas anuales, mensuales y diarias.
   - Gas: el cual es medido en termia, con gráficas anuales, mensuales y diarias
   - Emisiones de carbón: medido en toneladas, con gráficas anuales, mensuales y diarias.
   - Análisis: gráfica que indica el consumo estimado según la categoría. Las categorías son: refrigeración, iluminación, calefacción, etcétera.

Postcondiciones:	El sistema se mantiene en el mismo estado.

3.2.8. Administrar plan de ahorro

El plan de ahorro es un conjunto de recomendaciones que el usuario se ha comprometido a llevar a cabo en su edificio. La administración de esta lista es de vital importancia para el sistema PEAT puesto que permite al usuario tomar acciones concretas para reducir sus costos de energía.

Caso de Uso:	**Administrar plan de ahorro**
Descripción:	El sistema debe permitir la administración del plan de ahorro del edificio, el cual consiste en un conjunto de recomendaciones que el usuario sea ha comprometido a realizar o bien que ya ha terminado para reducir su gasto energético.
Actor:	Usuario PyME

Requerimientos no funcionales:	Las actualizaciones a la página deben hacerse en forma asíncrona y sin necesitar una recarga completa de la página.

Flujo normal:

1. El usuario no ha agregado ninguna recomendación a su plan

   a) Las tres mejores recomendaciones son desplegadas dentro del plan de ahorro.

   b) El sistema elige estas recomendaciones por medio de la siguiente lógica:

   - Son ordenadas según su período de recuperación (definido por el ahorro anual en dólares / costo en dólares, por adelantado).
   - Recomendaciones con costo inicial cero (por ejemplo, acciones basadas en comportamiento) se consideran que tienen un reembolso de dos años.
   - Las recomendaciones que han sido marcadas como completadas o que no aplican no son parte de la selección para el plan de ahorro.

2. El usuario a agregado una o más recomendaciones a su plan de ahorro.

   - La mejor recomendación es desplegada debajo del componente PPW.
   - El sistema despliega una lista de las recomendaciones que el usuario a agregado a su plan.
     - El sistema debe indicar que la lista de recomendaciones es el plan de ahorro.
     - El usuario puede agregar más recomendaciones desde la lista de recomendaciones, cuando se añade una de estas solamente es visible en el plan de ahorro ya no debe de aparecer en la lista de recomendaciones
     - El usuario puede remover recomendaciones de su plan seleccionando «Remover del plan» en un menú desplegable al lado del nombre de la acción.
   - El usuario puede obtener más información al seleccionar una recomendación.
   - El usuario actualiza el estado de una recomendación a «Completada» o «Remover».
     - Las recomendaciones dentro del plan de ahorro solo pueden tener dos posibles estados: «Completada» y «Remover», el estado base de una recomendación agregada al plan es de «Añadida».
     - Las recomendaciones con estado de «Completada» permanecen en el plan de ahorro con la selección «Completada» desplegada al lado del nombre de la acción.

Postcondiciones:	Si el usuario realizó cualquier acción en una recomendación entonces el plan de ahorro es actualizado y reordenado.

3.3. Interfaz de usuario

Dados los requerimientos funcionales y no funcionales presentados en los casos de uso se encuentra que la interfaz de usuario debe tener las siguientes características:

- La interfaz debe ser simple y enfocada a la información que el usuario necesita.
- La interfaz debe permitir realizar la mayoría de sus acciones de forma asincrónica, sin necesidad de una recarga completa de la página.
- Toda interacción debe completarse en menos de un segundo.

Para cumplir estos requerimientos fue necesario una cantidad considerable de código Javascript en la parte del *frontend*, para realizar peticiones al servidor de forma asíncronas por medio de AJAX, y realizar las actualizaciones a la vista después de recibir la respuesta del servidor.

Para permitir el reuso de código para la implementación de funcionalidad como el *componente de ingreso progresivo* era necesario dar estructura a los componentes del *frontend*, para este fin se eligió utilizar la biblioteca *Backbone* pues permite definir una arquitectura MVC, es decir, permite definir modelos, vistas y controladores en la parte del *frontend*.

3.3.1. Descripción general y navegación

La interfaz de usuario en el sistema PEAT está compuesta por ocho páginas de las cuales tres son páginas de PG&E y las restantes son gestionadas por PEAT.

Todas las páginas del sistema muestran un menú de navegación que permite al usuario navegar directamente a las páginas «Plan de ahorro», «Recomendaciones» y «Perfil detallado del edificio».

La descripción general de las páginas es la siguiente:

1. Un usuario ingresa al portal de PG&E, luego a la página de ingreso del sistema *MyEnergy* de PG&E, en este sistema es donde los usuarios PyME realizan la administración de su perfil y cuentas con PG&E. En el tablero de *MyEnergy* se tiene una liga que invita al usuario a conocer sobre medidas para ahorrar, esta liga dirige al usuario al sistema PEAT (Ver Figura 3.3 1).

2. En el primer ingreso al sistema se dirige al usuario a la página «Crear perfil del edificio» (Ver Figura 3.3 2), aquí el usuario solo contesta las preguntas básicas sobre su edificio (Ver caso de uso *Crear perfil del edificio*). El usuario regresa a esta página cuando requiere agregar más edificios a su cuenta. Al finalizar la creación del perfil se redirige al usuario a la página de «Plan de ahorro».

3. La página de «Plan de ahorro» es la página principal del sistema PEAT, es decir, la página inicial (cuando el usuario cuenta por lo menos con un un perfil de edificio) al ingresar por medio de su tablero en *MyEnergy* (Ver Figura 3.3 3). El plan de ahorro presenta la siguiente funcionalidad al usuario:

- Parte superior: se proporciona al usuario capacidad para interactuar con la facturación de sus gastos de energía por medio de gráficas comparativas (Ver caso de uso *Ver historial de consumo*).
- Parte inferior: se muestra al usuario su plan de ahorro, es decir, las recomendaciones que se ha comprometido a realizar para reducir su gasto (Ver caso de uso *Administrar plan de ahorro*).

El usuario puede navegar a la página de «Detalle de recomendación» mostrada en su plan de ahorro.

4. En la página de «Recomendaciones» se tienen dos componentes (Ver Figura 3.3 4):
   - El componente de ingreso progresivo que invita al usuario a dar mas información sobre su edificio (Ver caso de uso *Obtener información del edificio*).
   - Una lista de recomendaciones para reducir sus costos de energía basados en el perfil de su edificio (Ver caso de uso *Administrar recomendaciones*).

   El usuario puede navegar a la página de «Detalle de recomendación» de las recomendaciones mostradas en la lista de recomendaciones.
5. La página de «Perfil detallado del edificio» (Ver Figura 3.3 5) es donde el usuario puede responder o actualizar las respuestas a las preguntas sobre su edificio (Ver caso de uso *Administrar perfil del edificio*).
6. La página de «Detalle de recomendación» (Ver Figura 3.3 6) muestra los pormenores sobre las medidas que se necesitan realizar para lograr reducir el consumo en el edificio (Ver caso de uso *Ver recomendación*).

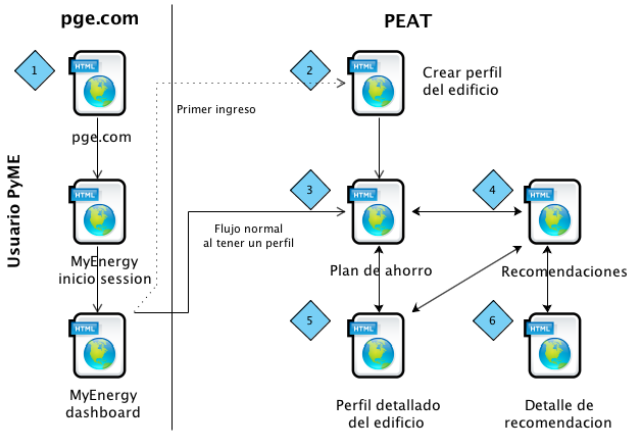

Figura 3.3: Diagrama de alto nivel de la navegación en el sistema PEAT.

3.3.2. Encabezado general

Todas las páginas del sistema PEAT comparten el mismo encabezado, como se muestra en la Figura 3.4, el cual tiene las siguientes secciones:

a) Componente de selección de edificios: este control muestra un menú desplegable con todos los edificios asociados al usuario.

b) Nombre del sistema: el nombre es «Chequeo de energía del negocio» (*Business Energy Checkup*), el nombre interno del sistema es PEAT.

c) Menú de navegación del sistema.

- Plan de ahorro (*Energy Plan*): dirige al usuario a la página de «Plan de ahorro».
- Formas de ahorrar (*Ways to Save*): dirige al usuario a la página de «Recomendaciones».
- Perfil de negocio (*Business Profile*): dirige al usuario a la página de «Perfil detallado del edificio».
- Tomar un recorrido (*Take a tour*): se muestra al usuario las diferentes páginas y secciones del sistema.

d) Título de la página actual.

e) Componente de potencial de ahorro: este control muestra en todo momento el potencial de ahorro del edificio y el ahorro que se obtiene por el plan de ahorro del edificio.

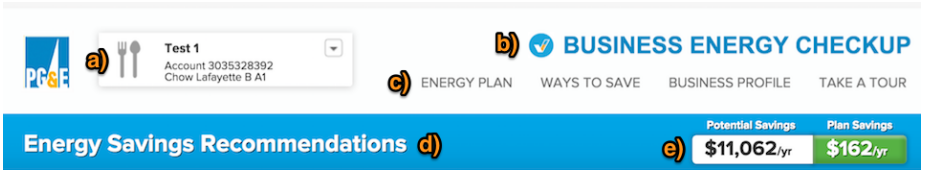

Figura 3.4: Encabezado principal del sistema PEAT.

3.3.3. Crear perfil del edificio

La página para «Crear perfil del edificio» es la página inicial del sistema cuando el usuario no tiene ningún perfil del edificio asociado, esta página también es usada cuando el usuario agrega más edificios a su cuenta.

Esta página tuvo varias iteraciones por la retroalimentación obtenida por las pruebas con los usuarios, en las primeras iteraciones la página requería que el usuario contestara toda la información requerida según el caso de uso *Crear perfil del edificio* como se muestra en la Figura 3.5, lo cual representaba mucho trabajo al usuario.

Para facilitar al usuario el ingreso de información se decidió implementar las siguientes mejoras:

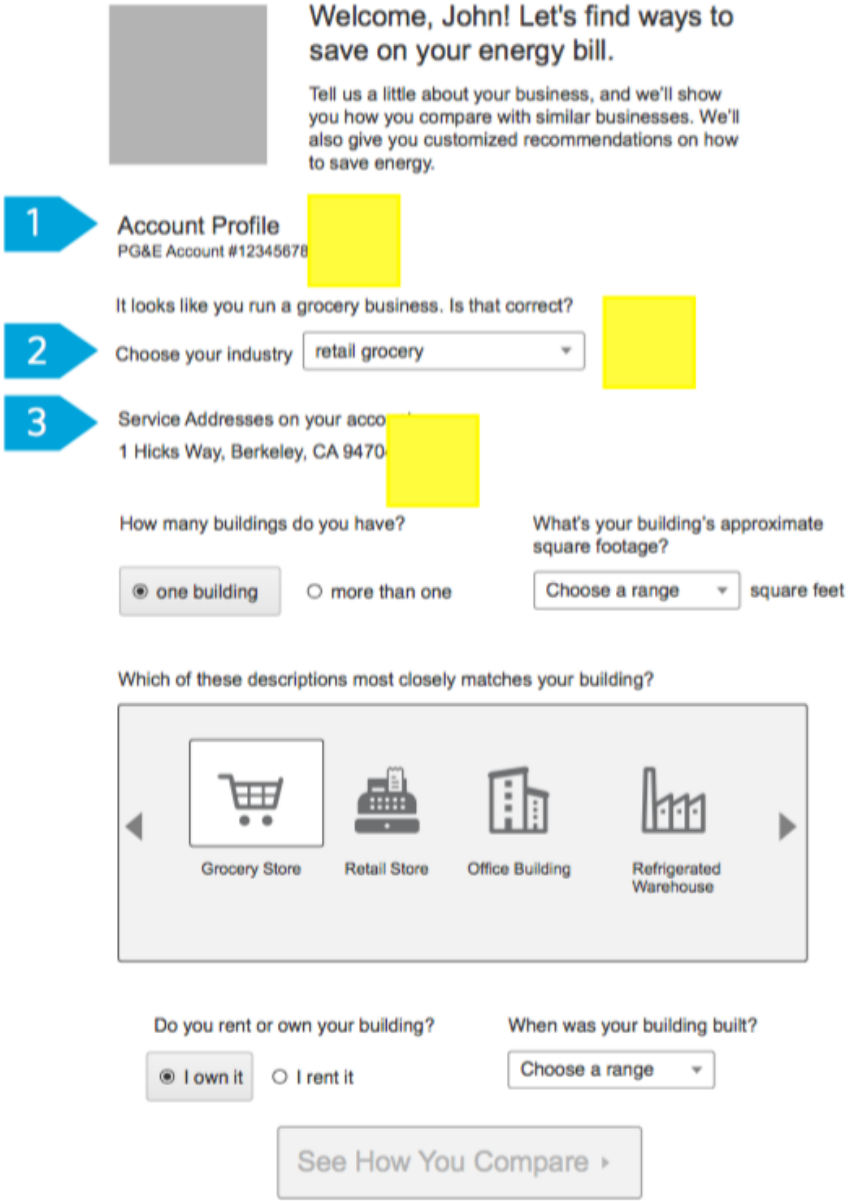

Figura 3.5: Versión inicial para crear un perfil de edificio.

1. Se redujo el número de opciones para el tipo de industria, inicialmente esta lista era numerosa lo que implicaba mas trabajo al usuario, se redujo a cinco opciones que son visibles inmediatamente.
2. Se dividió el ingreso de información en dos partes.

En la versión final el sistema inicialmente pide al usuario seleccionar el tipo de industria que mejor describe su negocio, las opciones son: comercial, industrial, agrícola, multifamiliar y otro como se puede ver en la Figura 3.6.

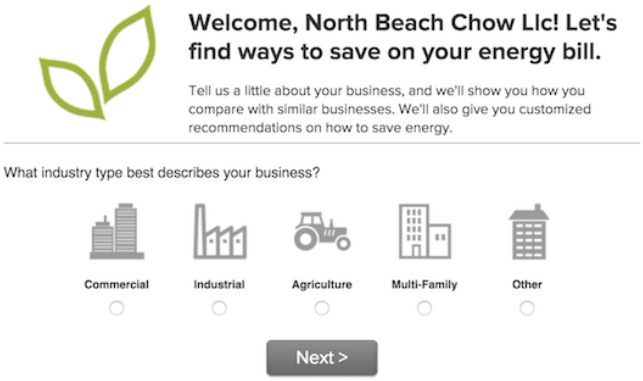

Figura 3.6: Se pide al usuario el seleccionar el tipo de industria de su negocio.

Después el sistema pide al usuario seleccionar su cuenta y dirección de servicio, el alias del edificio, el tipo de edificio y su área como se puede ver en la Figura 3.7.

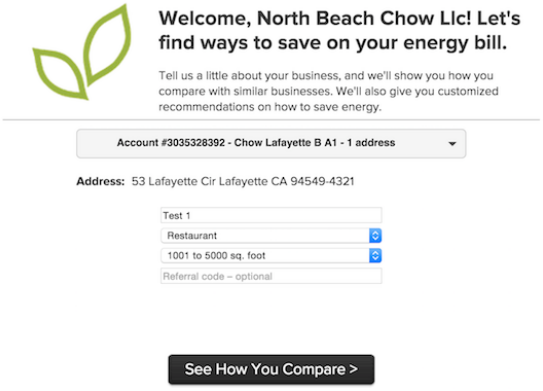

Figura 3.7: Se pide al usuario ingresar las características básicas de su edificio.

Al ingresar esta información el usuario es redirigido a la página de «Plan de ahorro» del perfil creado.

3.3.4. Plan de ahorro

La página de «Plan de ahorro» es la página principal del sistema PEAT dado que implementa la funcionalidad principal del sistema pues tiene la capacidad para interactuar con los datos sobre los gastos de energía y la administración del plan de ahorro del edificio.

La página está dividida en dos secciones principales, en la sección superior se da el manejo de las gráficas para visualizar los gastos de energía y en la sección inferior se tiene la administración del plan de ahorro.

Visualización

La sección superior consta de las siguientes partes:

- Menú de vistas: En este menú el usuario puede seleccionar que tipo de datos se despliega en las gráficas. Se tienen cuatro tipos de vista: gasto (*Spending*), electricidad (*Electricity*), gas (*Gas*), emisiones de carbono (*Carbon Emissions*) y análisis (*Energy Use*). También en este menú en la parte derecha se tiene un estimado del ahorro que se puede obtener en el edificio si se implementan las recomendaciones (Ver Figura 3.8).

Figura 3.8: Menú de vistas.

- Menú de rango de tiempo: Esta sección se encuentra enseguida del menú de vistas, aquí el usuario puede seleccionar entre los rangos de tiempo anual (*Annually*), mensual (*Monthly*) o diario (*Daily*) de la vista seleccionada, en ciertas vistas se tienen opciones extra en la parte derecha como por ejemplo agregar datos de clima o de periodos anteriores a la vista actual (Ver Figura 3.9).

Figura 3.9: Menú de rango de tiempo.

- Área de visualización: En esta sección se despliega los datos de consumo del edificio mostrando los datos y rango de tiempo que el usuario ha seleccionado en los dos menús anteriores (Ver Figura 3.10).

Para la implementación de las gráficas se utilizó el marco de trabajo Ext JS, este marco de trabajo está diseñado para implementar aplicaciones de tipo *SPA* por medio del lenguaje de programación Javascript. Ext JS implementa un gran número de controles de interfaz como paneles, barras de herramientas, pestañas, etcétera.

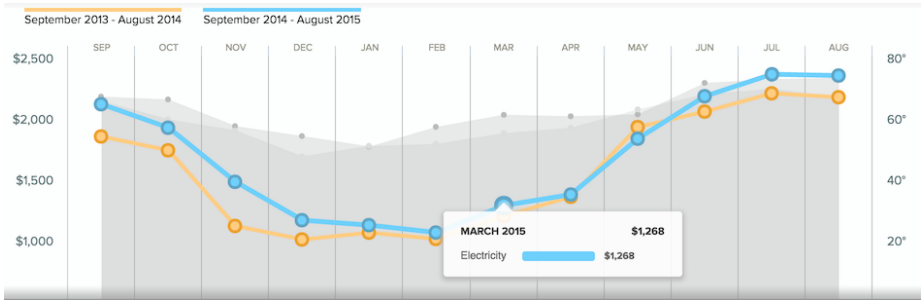

Figura 3.10: Visualización del gasto de electricidad.

Ext JS es una elección natural para el sistema PEAT dado que tiene un excelente paquete de graficación el cual permite presentar visualmente una gran cantidad de datos por medio de una amplia gama de gráficas: líneas, barras, circulares, etcétera. Otro punto a favor de Ext JS es que el equipo de programadores del *frontend* estaba ampliamente familiarizado con este marco de trabajo.

Dada la capacidad de Ext JS para crear interfaces se tenía el plan inicial de implementar toda la interfaz con este marco de trabajo sin embargo Ext JS tiene una gran desventaja ya que requiere de tiempo para inicializar sus componentes demasiado grande, (mas de un segundo), lo cual se contrapone a los requerimientos del cliente. Así se decidió hacer uso solamente del módulo de graficación de Ext JS y hacer la implementación de los controles necesarios para la interfaz desde cero haciendo uso de Backbone.

Para cada rango de tiempo se utiliza un tipo de gráfico y/o visualización diferente:

- Anual: se hace una comparativa entre el gasto del edificio en los últimos doce meses, el promedio de gasto de un edificio de características similares y el gasto de un edificio eficiente en su consumo energético. Para hacer estas comparaciones se utiliza gráficas de barras como se puede ver en la Figura 3.11.

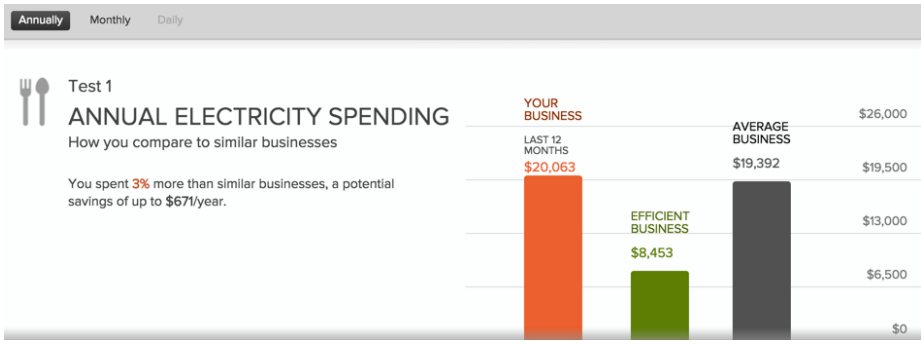

Figura 3.11: Visualización anual del gasto de electricidad.

- Mensual: se muestra una gráfica de líneas del gasto del edificio en los últimos 12 meses, además permite comparar con el gasto del año anterior como se puede observar en la Figura 3.12. Además se pueden agregar los datos sobre el clima durante el período que se está visualizando.

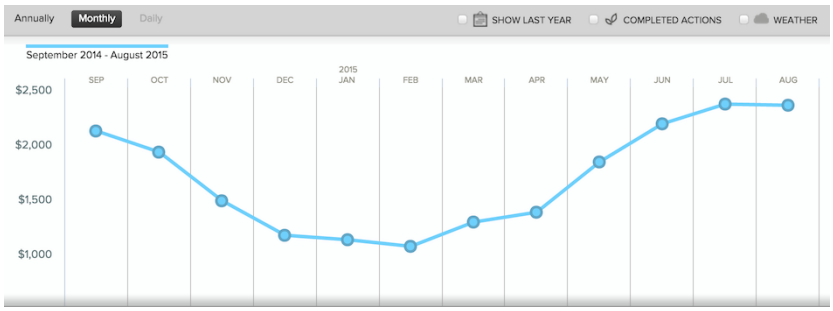

Figura 3.12: Visualización mensual del gasto de electricidad.

- Diario: Se muestra una gráfica similar a la vista mensual, pero en este caso se visualizan los últimos treinta días de gasto como se muestra en la Figura 3.13.

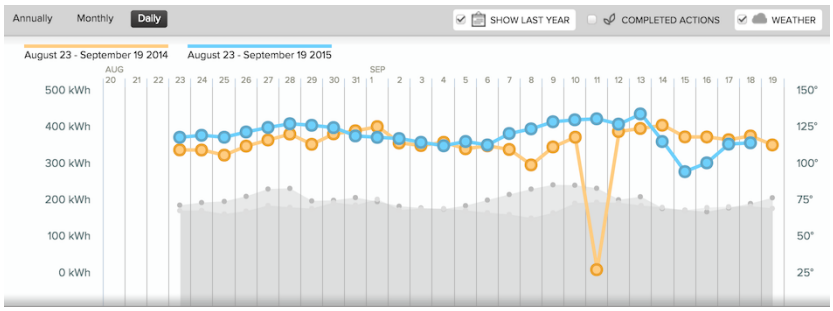

Figura 3.13: Visualización diaria del gasto de electricidad, con la comparativa al gasto del año pasado y datos del clima.

- Análisis: Se muestra un análisis del gasto del edificio en categorías por medio de una gráfica circular como se puede observar en la Figura 3.14, este análisis es realizado usando aprendizaje de máquina por lo que este análisis es solo una estimación.

Aunque las figuras solo muestran el caso del gasto en electricidad las visualizaciones son idénticas para el gasto en gas.

Figura 3.14: Visualización del análisis del gasto de electricidad.

Administración

La sección inferior como se muestra en la Figura 3.15 consta de las siguientes partes:

- Componente de ingreso progresivo: en este se despliega una pregunta sobre el edificio del usuario, las características de este componente se verán en detalle en la sección 3.3.5.
- Componente lista de recomendaciones: en este componente se despliega la lista de recomendaciones que forman el plan de ahorro del edificio, las características de este componente se verán a detalle en la sección 3.3.8.

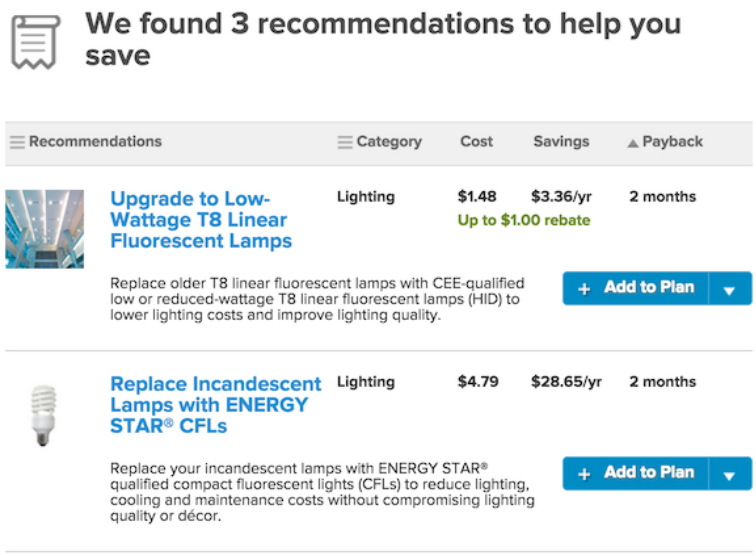

Figura 3.15: Plan de energía inicial con recomendaciones.

3.3.5. Recomendaciones

La página de «Recomendaciones» muestra al usuario una lista de sugerencias acerca de las medidas que se pueden realizar en el edificio para lograr disminuir el consumo energético del edificio. En esta lista se tienen todas las recomendaciones aplicables que no están ya en el plan de ahorro del edificio.

La página tiene dos secciones: la sección lateral contiene el componente de ingreso progresivo y en la sección principal se tiene el componente lista de recomendaciones.

Componente de lista de recomendaciones

El componente de lista de recomendaciones consta de las siguientes secciones:

- Encabezado: se muestran las características principales por las cuales el usuario puede ordenar la lista ya sea por nombre, categoría, costo, ahorro y recuperación de la inversión, como se puede ver en la Figura 3.16.

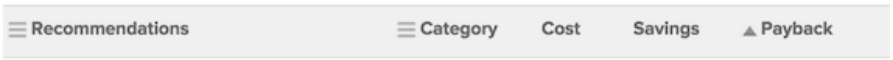

Figura 3.16: Encabezado de la lista de recomendaciones.

- Renglón: por cada recomendación en la lista se despliegan las características principales definidas en el encabezado, además se tiene un botón que permite al usuario cambiar el estado de la recomendación. Para cada recomendación se tiene las siguientes secciones como se puede ver en la Figura 3.17:

  a) Imagen representativa de la recomendación.
  b) Descripción de una línea de la recomendación.
  c) Categoría de la recomendación.
  d) Costo de implementar la recomendación.
  e) Ahorro que trae la recomendación en un año.
  f) Tiempo para recuperar la inversión, es decir, cuanto tiempo el ahorro obtenido al implantar la recomendación es igual o superior al costo de su implementación.
  g) Breve descripción de las medidas necesarias para implementar la recomendación.
  h) Componente añadir al plan, este componente permite al usuario definir el estado de la recomendación, es decir, si la recomendación es parte del plan de ahorro, ya fue implementada o no aplica al edificio.

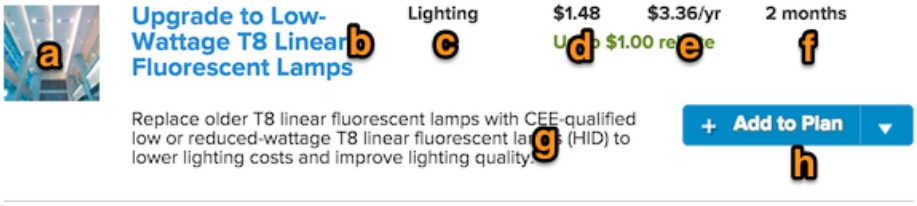

Figura 3.17: Las secciones de una recomendación.

Componente añadir al plan de ahorro

El componente «Añadir al plan» permite al usuario cambiar el estado de una recomendación, los estados posibles son:

- Por implementar: en este estado la recomendación no es parte del plan de ahorro del edificio, este hace referencia al estado inicial de las recomendaciones.
- Añadido: este estado indica que el usuario ha elegido implementar la recomendación y por lo tanto es parte del plan de ahorro del edificio.
- No aplicable: este estado indica que la recomendación no es aplicable al edificio.
- Completada: se refiere a la recomendación terminada la cual ya fue implementada con éxito en el edificio, ésta se considera parte del plan de ahorro.

Mediante la interacción con este componente el usuario modifica el estado de una recomendación. Este componente tiene dos formas de interacción:

- Presionando su parte izquierda: se realiza la acción de agregar la recomendación al plan de ahorro.
- Presionando su parte derecha: se despliega un menú de acciones secundarias que dependen del estado actual de la recomendación.

Como se indico anteriormente para realizar la transición del estado «Por implementar» al estado «Añadido» el usuario solo tiene que presionar el control en su parte izquierda como se puede ver en las Figuras 3.18 y 3.19.

Figura 3.18: Vista inicial del control de añadir al plan.

El sistema despliega una ventana como se puede ver en la Figura 3.20 preguntando al usuario una estimación del tiempo en que se podrá implementar las medidas propuestas por la recomendación que se esta agregando al plan de ahorro.

Figura 3.19: Vista cuando la recomendación a sido añadida al plan.

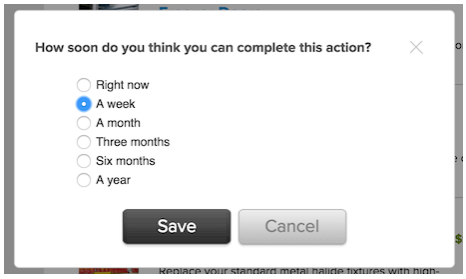

Figura 3.20: Al agregar una recomendación al plan el sistema pregunta el tiempo estimado para completar la recomendación.

Para realizar otras acciones sobre la recomendación el usuario presiona la parte derecha del componente en la parte donde se encuentra un triángulo que indica la presencia de un menú despegable. Estando la recomendación es un estado inicial «Por implementar» las acciones en el menu despegable son para indicar que se ha completado la recomendación (estado «Completada») o que no es aplicable (estado «No aplicable») como se puede observar en la Figura 3.21.

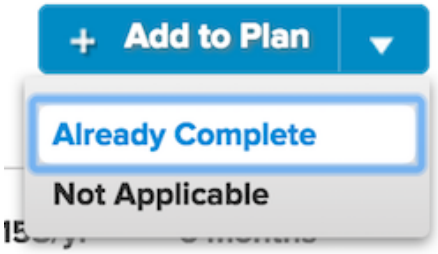

Figura 3.21: Las acciones de completar o indicar no aplicable en el menú despegable.

Cuando la recomendación se encuentra en el estado de «Añadido» las acciones en el menú despegable son las relacionadas con completar, remover o indicar que no es aplicable a la recomendación como se muestra en la Figura 3.22.

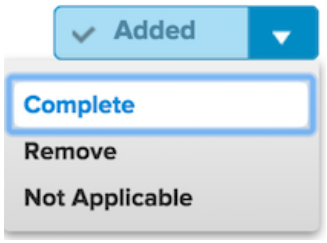

Figura 3.22: Las acciones de completar, remover o indicar no aplicable en el menú despegable.

3.3.6. Detalle de recomendación

La página de «Detalle de recomendación» describe de forma detallada la recomendación y las medidas que se tienen que implementar para lograr reducir el consumo energético del edificio.

La página está dividida en las siguientes secciones:

- Encabezado: se muestra una descripción breve de la recomendación y a su derecha se tiene el componente «Añadir al plan» como se muestra en la Figura 3.23.

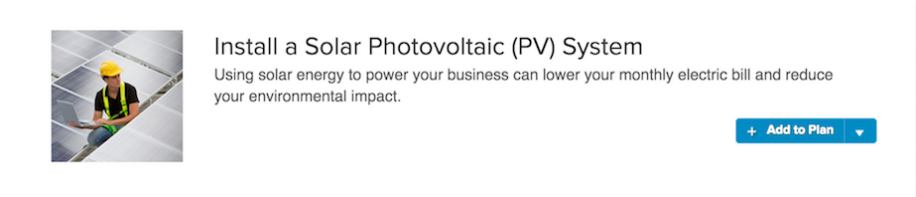

Figura 3.23: Encabezado de la recomendación para la instalación de un sistema de paneles solares.

- Descripción: se describe a detalle la información sobre las medidas necesarias para reducir el consumo energético del edificio.
- Preguntas: Como se puede observar en la Figura 3.24 después de la descripción se pueden tener una o varias preguntas al usuario para obtener mas datos con el fin de establecer una mejor estimación sobre los costos y ahorro esperados al completar la recomendación. En la sección lateral se despliega el resultado del cálculo estimado de costos y ahorro al completar la recomendación, estos datos son actualizados de forma asíncrona cuando el usuario contesta o cambia la respuesta a alguna de las preguntas presentadas en el contenido principal.

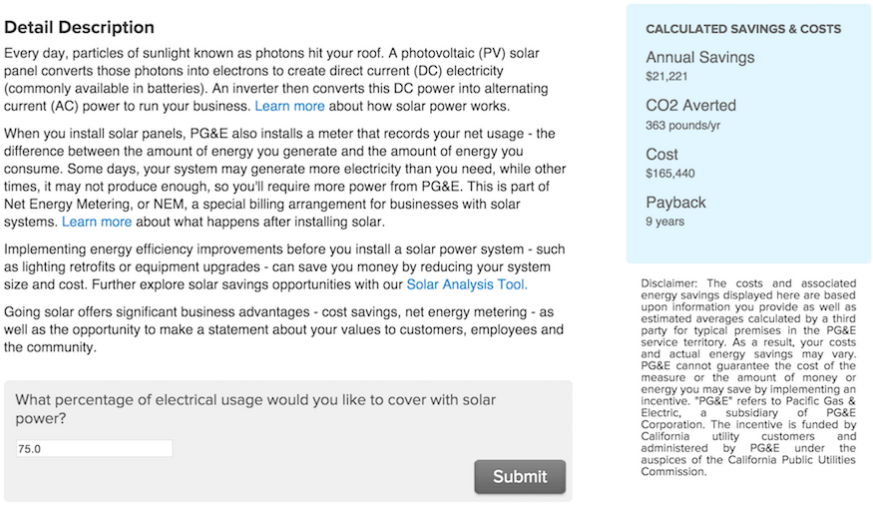

Figura 3.24: Descripción detallada de la recomendación, con la estimación de costo y ahorro en la sección lateral.

3.3.7. Perfil detallado del edificio

La página de «Perfil detallado del edificio» permite al usuario el ingreso o actualización de las características de su edificio por medio de un conjunto de preguntas y respuestas divididas en categorías, como se puede observar en la Figura 3.25.

La página se encuentra dividida en varias secciones[1]:

- Edificio (*Building*).
- Iluminación (*Lighting*).
- Calefacción, ventilación y aire acondicionado (*Heating, Ventilation and Air Conditioning* HVAC).
- Refrigeración (*Refrigeration*).
- Cocina (*Cooking*).
- Calefacción de agua (*Water-Heating*).
- Miscelánea (*Misc*).

Varias categorías como cocina y refrigeración solo son mostradas para ciertos tipos de edificios.

El usuario selecciona una categoría seleccionando el título de una categoría, para que esa sección presente las preguntas y respuestas asociadas a ésta.

Cada sección está dividida, como se muestra en la Figura 3.26, en las siguientes partes:

[1] Una sección por categoría

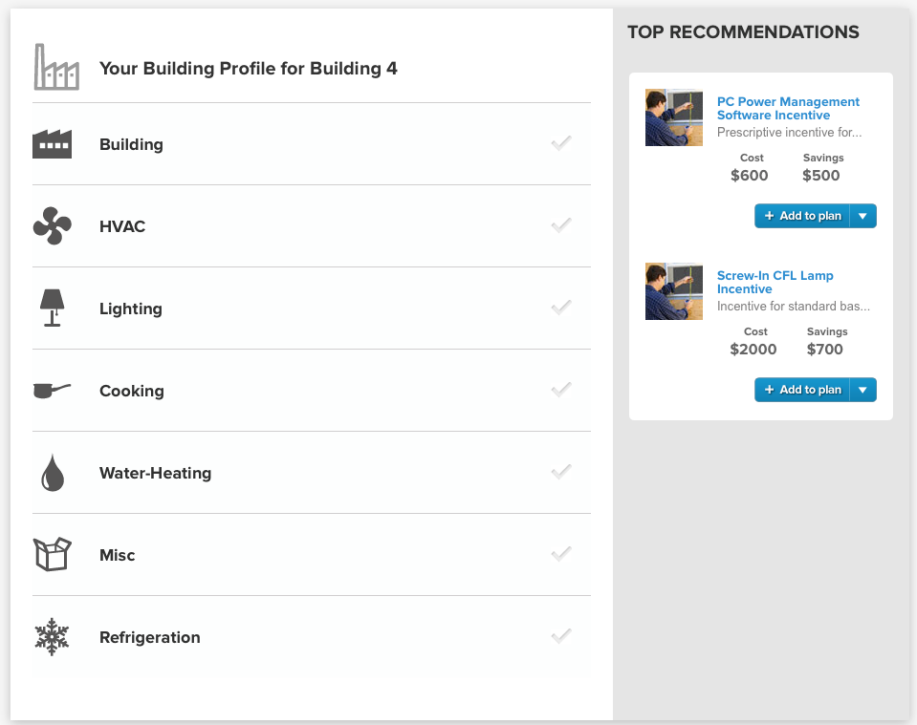

Figura 3.25: Las preguntas y respuestas sobre el edificio se organizan en categorías.

1. Encabezado de la sección: cada encabezado tiene un ícono característico, el nombre de la categoría y en la parte derecha un marcador indica si todas las preguntas de la sección han sido respondidas por el usuario (Ver Figura 3.26 1).

2. Preguntas y respuestas: las preguntas son breves y se refieren a una característica particular del edificio. La mayoría de las preguntas tiene respuestas definidas[2]. Las respuestas se presentan de las siguientes formas (Ver Figura 3.26 2):

   a) Menú desplegable: para preguntas que tienen mas de tres respuestas posibles excluyentes.

   b) Casillas de selección múltiple: para preguntas en que las respuestas pueden ser validas al mismo tiempo.

   c) Campo de texto limitado (solo un valor numérico): para las preguntas en que se necesita un valor numérico exacto.

   d) Botón de opción (con texto o imagen): para preguntas que tiene un número limitado de respuestas posibles. Se hace uso de imágenes para las preguntas más técnicas para facilitar al usuario el contestar la pregunta.

[2]Se cuenta solo con un reducido número de preguntas donde se espera el ingreso de un valor numérico

3. Botón siguiente: para progresar a la siguiente categoría (Ver Figura 3.26 3).

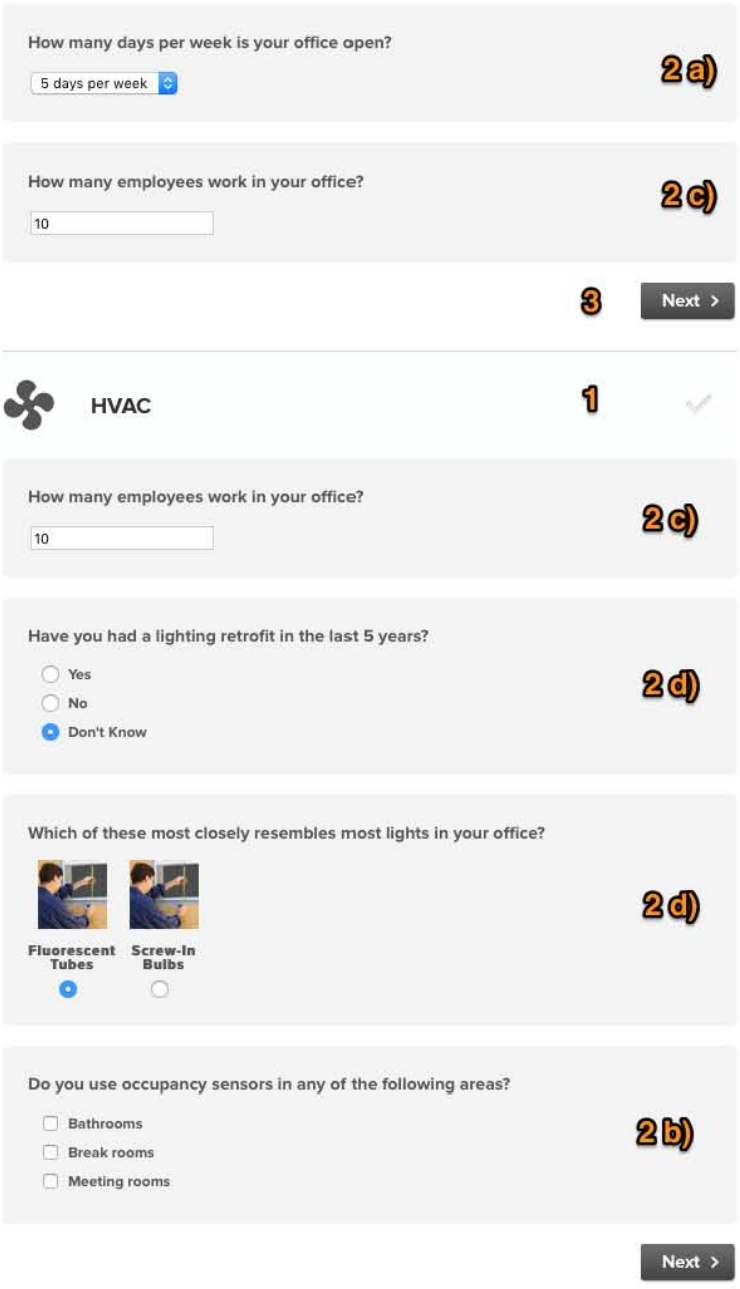

Figura 3.26: Las preguntas de las secciones Edificio y HVAC.

3.3.8. Componentes reusables

En PEAT se tienen varios componentes como el componente de ingreso progresivo o el componente para añadir al plan de ahorro cuya presencia es necesaria en varias de las páginas del sistema por la funcionalidad que proporcionan. Para permitir el reuso de código y diseño de estos componentes se decidió hacer uso de la biblioteca Backbone, dado que permite dar estructura (definiendo modelos, vistas y controladores) al implementar la funcionalidad requerida.

Los componentes que se implementaron son los siguientes:

- Componente de ingreso progresivo.
- Componente del potencial de ahorro.
- Componente para lista de recomendaciones.
- Componente para añadir al plan de ahorro.

Componente de ingreso progresivo

Este componente muestra al usuario una pregunta y sus posibles respuestas sobre el perfil del edificio. Este componente es usado en las páginas de «Plan de ahorro» y de «Recomendaciones».

El componente, como se puede ver en la Figura 3.27, tiene las siguientes partes:

a) Pregunta y respuesta: se despliega una sola pregunta y las respuestas se presentan según los tipos de respuesta expuestos en el perfil del edifico.

b) Botón saltar (*Skip*): permite al usuario omitir dicha pregunta mostrada y pasar a la siguiente pregunta sin contestar del perfil de su edificio.

c) Botón siguiente (*Next*): permite al usuario enviar la respuesta seleccionada de la pregunta mostrada.

d) Liga al perfil detallado: permite al usuario navegar al perfil detallado de su edificio.

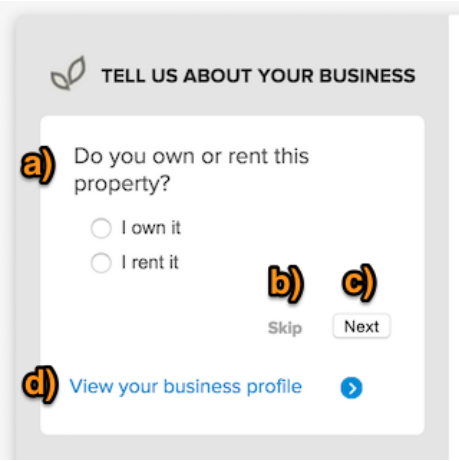

Figura 3.27: Componente de ingreso progresivo.

Componente del potencial de ahorro

Este componente permite al usuario conocer el potencial de ahorro del edificio y su ahorro esperado establecido en su plan de ahorro.

Componente lista de recomendaciones

Este componente lista las recomendaciones que se pueden realizar en un edificio para reducir sus gastos energéticos, la descripción detallada de este componente se encuentra en la Sección 3.3.5.

Componente para añadir al plan de ahorro

Este componente permite modificar el estado de una recomendación, la descripción detallada de éste se encuentra en la Sección 3.3.5. Este componente es usado en las páginas de «Plan de ahorro», «Recomendaciones» y «Detalle recomendación».

3.4. Biblioteca Bezel

Para lograr la integración entre el sistema PEAT y el servicio web C3, el API proporcionado por el servidor C3, se ha implementado una biblioteca con el nombre de **Bezel**.

PEAT hace uso del marco de trabajo Rails, el cual a su vez hace uso del patrón de arquitectura MVC. En Rails los modelos por lo general hacen uso de la biblioteca ActiveRecord la cual permite implementar modelos basados en tablas de bases de datos relacionales. En PEAT los datos son proporcionados por el servicio web C3 el cual es administrado por el servidor C3.

Dentro del marco de trabajo Rails también se tiene la biblioteca ActiveResource la cual implementa el caso de uso en que los datos son provistos por un servicio web tipo RESTful.

En una primera versión se trató de realizar la integración de PEAT con el servicio web C3 haciendo uso de esta biblioteca, sin embargo se tenían las siguientes dificultades:

- Presupone que el servicio web hace uso de todos los métodos HTTP (GET, POST, UPDATE, DELETE) para definir las acciones básicas sobre un recurso.
- No permite la definición de asociaciones entre recursos.

Aunque el servicio web C3 es de tipo RESTful, éste no hace uso de los métodos HTTP para realizar acciones sobre los recursos del servicio. El servicio web C3 realiza todas sus acciones usando el método HTTP POST.

Otra dificultad era que la biblioteca ActiveResource no da ninguna facilidad para definir asociaciones entre los modelos lo que implicaba el tener que implementar una gran cantidad de código dado el gran numero de asociaciones presentes entre los tipos definidos por el servicio web C3.

Dada las dificultades descritas en los párrafos anteriores el hacer uso de ActiveResource no era una solución viable, sin embargo sirvió como una primera aproximación para implementar la biblioteca Bezel y así proporcionar acceso al servicio web C3.

Cabe recordar que el servidor C3 está implementado en dos lenguajes de programación, la gran mayoría implementado con Javascript, mientras que el núcleo y funciones criticas están implementadas en Java. El servidor C3 puede enviar tanto XML como JSON como respuesta a las peticiones realizadas al servidor[3].

Bezel permite la reconstrucción automática de los tipos definidos por el servidor C3 en Javascript a modelos compatibles con el marco de trabajo Rails y el lenguaje de programación Ruby. Esta reconstrucción automática permite evitar desfases entre la definición de los tipos entre los dos sistemas y agiliza el desarrollo del sistema PEAT. Esta reconstrucción automática se hace por medio de un lenguaje de dominio específico que permite implementar modelos basados en tipos del servicio web C3 aprovechando que este servicio es tipo RESTful y las capacidades de metaprogramación de Ruby.

[3]La representación en JSON es la más utilizada dado que el sistema de tipos que se usa en el sistema esta definido en Javascript.

Bezel tiene dos modos principales de uso:

- Como un cliente REST con ciertas adecuaciones para una mejor integración con el API del servidor C3.
- Como un lenguaje de dominio específico interno para Ruby para lograr la implementación automatizada de tipos y sus asociaciones del servicio web C3.

3.4.1. Servicio web C3

El servicio web C3 es de tipo RESTful, es decir, cada recurso o tipo tiene una única dirección URL asociada. Las direcciones URL tienen la siguiente forma:

POST /api/<*version*>/<*tenant*>/<*module*>;<*tag*>/<*type*>?action=<*action-name*>

- *version*: indica la versión del servicio a la cual se está accediendo, el sistema PEAT hace uso de la versión 1.0.
- *tenant*: se refiera al producto o cliente, esto es porque cada producto y/o cliente tiene cierta funcionalidad específica. Para PEAT se tiene «peat» y «c3».
- *module*: indica el módulo al que se está accediendo, un conjunto de tipos se definen en un módulo en particular. Para el sistema PEAT el módulo más usado es el módulo «peat», haciendo uso de los módulos «billing» y «structure».
- *tag*: indica el tipo de ambiente, es decir, si el servicio web C3 es para ambiente de pruebas, control de calidad, producción, etcétera. *peatprod* y *peatqa* son algunas de las etiquetas que se usan en PEAT.
- *type*: se refiere al tipo al que se quiere acceder, por ejemplo «Building», «Location» y «BuildingEnergyConservationOption» son algunos de los tipos utilizados en PEAT.
- *action-name*: indica la acción que se quiere realizar sobre el tipo indicado anteriormente. En este caso «fetch», «upsert» y «remove» son algunas de las acciones que se pueden realizar.

Como se comentó anteriormente el servicio web C3 no es totalmente de tipo RESTful, principalmente porque utiliza del mismo método HTTP POST para todas las acciones, indicando por medio del «action-name» la acción a realizar. Para la implementación de Bezel solamente se espera que el servicio web tenga direcciones únicas para cada tipo definido.

Muchas acciones reciben ciertos parámetros los cuales son almacenados dentro del cuerpo de la petición POST, por lo que el único parámetro que se envía por medio de la URL es el nombre la acción a realizar.

Tipos del servicio web C3

Del servicio web C3 se manejaron los siguientes tipos (Ver Figura 3.28):

- User: este tipo representa un cliente de PG&E, solamente contiene información básica como el nombre, código postal y su correo electrónico.
- Account: este tipo representa uno o más contratos de servicio, también contiene información sobre la industria a la que pertenece el negocio del usuario.
- Building: este tipo representa un edificio y contiene las características básicas de un edificio: tipo, tamaño y antigüedad.
- Location: este tipo representa una dirección de servicio, es decir, una ubicación física donde se prestan los servicios.
- BuildingProfileAnswer: este tipo representa las respuestas dadas por el usuario o sus valores por defecto para las características de un edificio, tiene siempre una pregunta asociada.
- BuildingQuestion: este tipo representa una pregunta asociada a una característica de un edificio, tiene una o varias respuestas válidas asociadas.
- ValidAnswer: este tipo representa una respuesta válida asociada a una pregunta sobre una característica de un edificio, cabe recordar que la mayoría de las preguntas tiene un numero limitado de respuestas válidas.
- ProductCategory: este tipo representa las categorías en las que se clasifican las recomendaciones, preguntas y opciones de conservación de energía. Iluminación y calefacción son algunas de las categorías que se tienen definidas.
- EnergyConservationOption: este tipo representa una medida o opción de conservación de energía que se puede implementar en un edificio para lograr disminuir su consumo energético. En este tipo contiene la información como el nombre y descripción que se despliega al usuario final y también contiene la información sobre el costo, ahorro y recuperación de la inversión.
- Recommendation: este tipo representa la implementación de una opción de conservación en relación a un edificio, es decir, contiene la información de costo, ahorro y recuperación de la inversión para un edificio en particular en base a las características del edificio.
- RecommendationFAQ: este tipo contiene las preguntas frecuentes y sus respuestas en relación a la implementación de una recomendación.
- ChartData: este tipo representa toda la información de consumo obtenida de los medidores inteligentes instalados en las direcciones de servicio.

Los tipos «User», «Account», «Building» y «Location» ya estaban completamente definidos por el servicio web C3, los demás tipos fueron modelados en coordinación con el grupo de modelado del servidor C3.

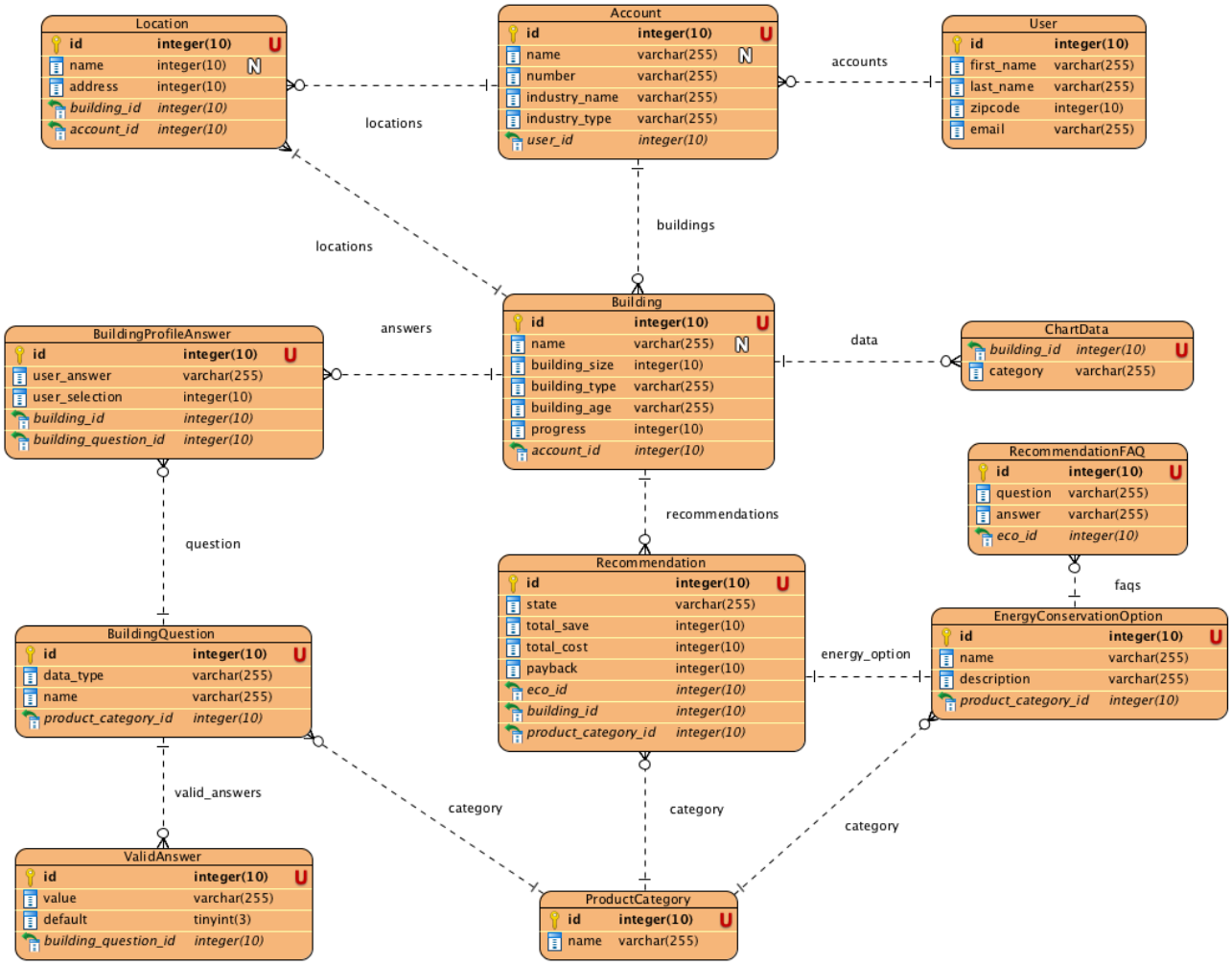

Figura 3.28: Diagrama tipo ER de los tipos del servicio web C3 utilizados para PEAT

3.4.2. Arquitectura general

La biblioteca `Bezel` está conformada por cinco clases y cinco módulos, siendo las clases principales `Bezel::Client` y `Bezel::Base`.

En el lenguaje de programación Ruby se tiene el concepto de módulo (`module`), el cual es mas que una agrupación de objetos bajo un único nombre. Los objetos pueden ser constantes, métodos, clases y otros módulos.

Los módulos tienen dos tipos de usos (1) se pueden utilizar un módulo como una manera conveniente de agrupar objetos o (2) se pueden incorporar los objetos del módulo a una clase haciendo uso de `include` o `extend`.

```
1  module Trig
2    PI = 3.14159
3    def Trig.sin(x)
4      # ...
5    end
6
7    def Trig.cos(x)
8      # ...
9    end
10 end
11
12 puts Trig.sin(Trig::PI/4) # Imprime: 0.7071067811865475
```

El ejemplo anterior muestra el uso de los módulos como un espacio de nombres, en la línea 2 se define la constante `PI` y en las líneas 3 y 7 se definen los métodos `sin` y `cos`, estos métodos son usados fuera del módulo por medio del nombre del módulo como ocurre en la línea 12, en esta línea se hace referencia tanto a la constante `PI` como al método `sin`.

```
1  module Boolean
2    def boolean?
3      true
4    end
5
6    def to_bool
7      return true if self == true || self =~ (/(true|t|yes|y|1)$/
       i)
8      return false if self == false || self.empty? || self =~ (/(
       false|f|no|n|0)$/i)
9      raise ArgumentError.new("invalid value for Boolean: \"#{
       self}\"")
10   end
11 end
12
13 module RandomString
14   def random
15     (0...8).map { (65 + rand(26)).chr }.join
16   end
17 end
18
```

```
19 class String
20   include Boolean
21   extend RandomString
22 end
23
24 puts "no".to_bool     # false
25 puts "false".boolean? # true
26 puts String.random    # cadena aleatoria de tamaño 8
```

Ahora en este ejemplo se muestra el uso de los módulos para incorporar métodos de clase o de instancia. En las líneas 1-11 se define el módulo `Boolean` el cual define dos métodos `boolean?` y `to_bool`, mientras que en las líneas 13-17 se define el módulo `RandomString` el cual a su vez define un método `random`. En las líneas 19-22 se permite modificar cualquier clase en tiempo de ejecución aunque esta clase sea parte de la biblioteca estándar, así en la línea 20 se hace uso de `include` para agregar los métodos del módulo dado como parámetro como métodos de instancia, el uso de estos métodos se puede ver en las líneas 24-25. Finalmente en la línea 21 se hace uso de la declaración `extend` para agregar los métodos del módulo dado como métodos de clase como se puede ver en la línea 26.

Por lo anterior se tiene que el uso de módulos es una parte importante para organizar y compartir funcionalidad en el lenguaje de programación Ruby.

Para la biblioteca `Bezel` tenemos el módulo `Bezel` que contiene a todas las clases y módulos de la biblioteca. Además se tienen cinco submódulos que permiten organizar de forma coherente la funcionalidad de la biblioteca.

Las clases y módulos que conforman la biblioteca son (Ver Figura 3.29):

- `Bezel`: módulo principal que contiene a los demás submódulos y clases de la biblioteca.
- `Bezel::Config`: módulo que define los métodos y valores válidos para configurar el cliente con el servicio web C3.
- `Bezel::Client`: clase que define un cliente para usar el servicio web C3.
- `Bezel::Target`: clase que representa un tipo en un contexto específico (*tenant*, *tag*, *module*).
- `Bezel::Action`: clase que representa una acción sobre un tipo representado por un `Bezel::Target`.
- `Bezel::Base`: clase abstracta que permite definir un modelo en Ruby con base en un tipo del servicio web C3.
- `Bezel::Connections`: módulo que define las acciones básicas y de búsqueda sobre un modelo.
- `Bezel::Associations`: módulo que define las asociaciones entre modelos.
- `Bezel::CacheBase`: módulo que redefine las acciones básicas y de búsqueda sobre un modelo haciendo uso de un repositorio de memoria `Bezel::Cache`.

- `Bezel::Cache`: clase que define un API para realizar operaciones de lectura/escritura sobre un repositorio de memoria.

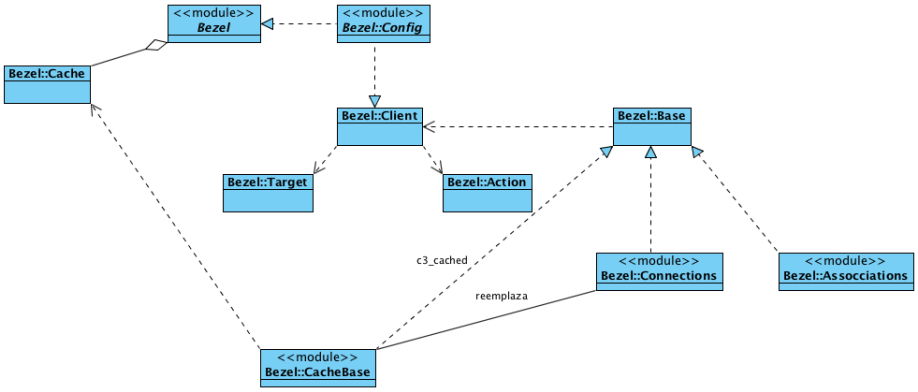

Figura 3.29: Diagrama de clase de la biblioteca Bezel

Dependencias

En la implementación de `Bezel` se hace uso de las siguientes bibliotecas de apoyo:

- moneta: proporciona un API uniforme para realizar operaciones de lectura y escritura en un repositorio de memoria.
- faraday: proporciona un API uniforme para realizar peticiones HTTP.
- hashie: proporciona un conjunto de métodos que extienden la funcionalidad de la clase `Hash` en Ruby.
- activemodel: proporciona interfaces para modelos que deben operar sobre Rails.

3.4.3. `Bezel`

El módulo `Bezel` no solo sirve como el espacio de nombres principal para la biblioteca, pues a su vez implementa varios métodos auxiliares para facilitar la configuración y creación de clientes para acceder al servicio web C3.

```
1 module Bezel
2   extend Config
3
4   class << self
5     def new(options={})
6       Bezel::Client.new(options)
7     end
8
9     def invoke(*args)
```

```
10        client.invoke(*args)
11      end
12
13      def client
14        @@client ||= Bezel::Client.new()
15      end
16
17      def context(tenant, tag, auth_token = nil)
18        # ...
19      end
20    end
21  end
```

En el código anterior se tiene la implementación abreviada de `Bezel`, en la línea 2 por medio de `extend` se añaden varios métodos de clase para configurar la biblioteca. En las líneas 4-20 se definen los métodos auxiliares más importantes:

- `new` (líneas 5-7): regresa una nueva instancia `Bezel::Client`.
- `invoke` (líneas 9-11): realiza una petición al servicio web C3 por medio del cliente asociado al módulo `Bezel`.
- `client` (líneas 13-15): usando una variable de clase se inicializa o reusa una instancia `Bezel::Client`. Este método permite reutilizar un mismo cliente y conexión al servidor C3 evitando el costo de establecer un nuevo cliente y en establecer una conexión HTTP con el servidor C3.
- `context` (líneas 17-19): permite cambiar el contexto de las peticiones que se realizan dentro de un bloque, se regresa al contexto original al finalizar de ejecutar el código dentro del bloque.

`Bezel::Config`

El módulo `Bezel::Config` define los atributos que permiten configurar el comportamiento de la biblioteca y de los clientes que se obtienen por medio de ella. Este módulo se usa tanto en `Bezel` para definir una configuración global de la biblioteca como en `Bezel::Client` para tener también una configuración local por cliente.

```
1  module Bezel
2    module Config
3
4      VALID_OPTIONS_KEYS = [
5              :auth_token,
6              :base_uri,
7              :cache,
8              :user,
9              :password,
10             :tenant,
11             :tag
12             # ...
```

```
13      ]
14
15      DEFAULT_VALUES = {
16              :auth_token => ENV['BEZEL_AUTH_TOKEN'],
17              :base_uri => ENV['BEZEL_BASE_URI'] || "http://
        localhost:8080",
18              :cache => ENV['BEZEL_CACHE'] || false,
19              :user => ENV['BEZEL_USER'],
20              :password => ENV['BEZEL_PASSWORD'],
21              :tenant => ENV['BEZEL_TENANT'] || 'c3',
22              :tag => ENV['BEZEL_TAG'] || 'c3',
23              # ...
24      }.freeze
25
26      attr_accessor *VALID_OPTIONS_KEYS
27
28      # ...
29
30      def self.extended(base)
31        base.reset
32      end
33
34      def reset
35        VALID_OPTIONS_KEYS.each { |k| send("#{k}=",DEFAULT_VALUES
        [k]) }
36        self
37      end
38    end
39  end
```

En el módulo `Config` se definen dos constantes:

- `VALID_OPTIONS_KEYS` (líneas 4-13): un arreglo de símbolos que indica que atributos son válidos.
- `DEFAULT_VALUES` (líneas 15-24): una tabla *hash* donde se indican los valores por defecto de los atributos válidos.

Haciendo uso del método `attr_accessor` en la línea 26 se definen por cada símbolo en `VALID_OPTIONS_KEYS` un atributo de instancia con sus correspondientes métodos de acceso y escritura. El método `attr_accessor` es un ejemplo del uso de metaprogramación en Ruby, dado que con base en el símbolo dado, se genera un atributo y sus métodos de acceso.

En la línea 30 se tiene el método `extended` el cual es un *callback* que es llamado cuando un módulo es extendido, es decir, es utilizado como parámetro al método `extend`. Para el módulo `Config` se tiene que cuando el módulo es extendido, como es en el caso de `Config` dentro del módulo `Bezel`, se llama al método `reset` para inicializar todos los atributos definidos en el módulo con sus valores por defecto. El método `extended` es solo uno de varios métodos *callback* que proporciona Ruby para la metaprogramación.

Otro método muy importante para la metaprogramación es `send` el cual es usado en la línea 35 para inicializar los atributos definidos por `Config`, el método `send` invoca el método identificado con un símbolo o cadena dada, pasando los argumentos posteriores como argumentos a la invocación del método. Haciendo uso de `send` y de las dos constantes definidas en el módulo se puede inicializar cada uno de los atributos.

3.4.4. `Bezel::Client`

La clase `Client` representa un cliente que permite realizar peticiones al servicio web C3, por medio del módulo `Config` se permite configurar cada cliente instanciado.

```
1  module Bezel
2    class Client
3      include Config
4
5      # ...
6
7      def conn
8        @conn ||= Faraday.new(url: @base_uri) do |faraday|
9          # Configuracion cliente HTTP
10       end
11     end
12
13     def invoke(mod, typ, action_name, params={})
14       target = Target.new(@tenant, @tag, mod, typ)
15       action = Bezel::Action.new(target, action_name, params)
16
17       endpoint = "/api/#{@api_version}/#{target.tenant}/#{
       target.module};#{target.tag}/#{target.type}?action=#{
       action_name}"
18       body = JSON.generate(params)
19
20       # ...
21
22       opts = {
23         :headers => {
24           "content-type" => "application/json",
25           "user-agent" => "ruby-bezel",
26           "Accept-Encoding" => "gzip",
27           "Accept" => "application/json"
28         },
29         :body => body,
30       }
31
32       addCredentials(opts)
33
34       # ...
35
36       begin
37         r = conn.post(endpoint, opts) do |req|
```

```
38            req.options[:timeout] = Bezel.timeout
39            req.options[:open_timeout] = 10
40          end
41        rescue Faraday::Error::TimeoutError => timeout_error
42          # ...
43          raise Bezel::TimeoutError.new(action)
44        end
45
46        # Manejo de errores
47
48        action
49      end
50    end
51  end
```

En la línea 3 se tiene un `include` que añade las constantes y atributos para configurar al cliente. En las líneas 7-13 se define el método `conn` que inicializa una conexión HTTP según la configuración definida para el cliente y pidiendo que las respuestas mandadas por el servidor sean de tipo JSON.

En las líneas 13-49 se tiene la implementación del método `invoke`, en la línea 14 se define un *target* de tipo `Bezel::Target` usando tanto el contexto del cliente, es decir el *tenant* y *tag* mas la información del *module* y *type* proporcionados en los argumentos. En la línea 15 se define una acción de tipo `Bezel::Action` que representa una acción sobre un tipo con sus parámetros.

En la línea 17 se define la URL a la cual se hará la petición al servicio web C3, haciendo uso de la información de los objetos previos, se convierte los parámetros de la acción al formato JSON para ser enviados en el cuerpo de la petición POST al servicio web.

En las líneas 22 a 30 se definen los encabezados y cuerpo de la petición POST al servicio web. Finalmente en la línea 37 se realiza la petición POST por medio de la conexión creada por el método `conn` y con los parámetros definidos anteriormente.

Posteriormente el resultado de la petición es almacenado en el objeto `action` y este objeto es regresado como resultado de la invocación del método `invoke`.

3.4.5. `Bezel::Base`

`Bezel::Base` es una clase abstracta la cual define la mayor parte de su funcionalidad en base al nombre de la clase que hereda de ella. Es decir no se hace uso directo de `Bezel::Base` si no que al heredar de esta clase abstracta se obtiene funcionalidad para crear modelos basados en tipos provenientes del servicio web C3.

```
1 class Recommendation < Bezel::Base
2   set_c3_type :BuildingEnergyConservationOption
3
4   has_one :energyConservationOption, EnergyConservationOption
5 end
6
7 rec = Recommendation.find(1)
8 rec.totalCost
```

El código anterior ejemplifica el uso de `Bezel::Base` para definir el modelo por medio de herencia, en la línea 1 se define `Recommendation`. En la línea 2 se tiene el método de clase `set_c3_type` que permite al modelo definir a que tipo del sistema web C3 está ligado, en este caso el modelo esta acotado al tipo de nombre `BuildingEnergyConservationOption`. En la línea 4 se define que el modelo tiene una asociación con el modelo `EnergyConservationOption` en forma única por medio del método `has_one`.

Ya definido el modelo en la línea 7 se hace la búsqueda de una recomendación mediante su identificador único por medio del método `find`. En la línea 8 se obtiene el costo total de la recomendación buscada anteriormente por medio de `totalCost`. Los métodos `set_c3_type`, `has_one` y `find` son definidos por `Bezel::Base` pero el método `totalCost` es definido en tiempo de ejecución por medio de la metaprogramación.

Métodos de clase

Los métodos de clase que se definen en `Bezel::Base` son necesarios para definir métodos de configuración del modelo y para realizar operaciones de búsqueda sobre el modelo.

```
1  module Bezel
2    class Base
3      class << self
4        attr_reader :cache_type
5
6        def config_option(name, default)
7          singleton_class.instance_eval do
8            define_method(name) do
9              instance_variable_get(:"@#{name}") || default
10           end
11
12           attr_writer :"#{name}"
13           alias_method :"set_#{name}", :"#{name}="
14         end
15
16         define_method(name) do
17           self.class.send(name)
18         end
19       end
20
21       def find(*arguments)
22         # ...
23       end
24
25       def invoke(action, params)
26         Bezel.invoke(c3_module, c3_type, action, params)
27       end
28
29       def inherited(klass)
30         klass.config_option :c3_type, klass.model_name
```

```
31          klass.config_option :c3_module, 'peat'
32          klass.config_option :c3_include, nil
33          klass.c3_cached(false)
34        end
35
36        def c3_cached(flag = :local, opts = {})
37          if flag && Bezel.cache
38            include Bezel::CacheBase
39            @cache_type = flag
40            @cache_opts = opts
41          end
42        end
43      end
44    end
45  end
```

Los métodos principales son:

- `config_option` (líneas 6-19): este método permite la definición de nuevos atributos sobre el modelo en tiempo de ejecución por medio del uso de método `define_method` que permite crear métodos en tiempo de ejecución, con este método y de `attr_writer` se definen tanto atributos de clase como de instancia. El método toma dos parámetros el primero es el nombre de la configuración a crear y el segundo es el valor por defecto de la configuración.

- `find` (línea 21): este método permite realizar búsquedas sobre el modelo por medio de un identificador, también permite realizar búsquedas más avanzadas por medio de parámetros como `filter`.

- `invoke` (líneas 25-27): este método permite realizar un acción sobre el modelo, como se puede ver en la línea 26 se hace uso del método `invoke` del módulo `Bezel` haciendo uso del contexto del modelo, es decir, su módulo y tipo correspondiente.

- `inherited` (líneas 29-34): este método es parte de API de metaprogramación de Ruby, el método es invocado cuando una subclase de `Bezel::Base` es creada. La subclase es pasada como el parámetro `klass` y por medio del método `config_option` se definen tres configuraciones:

  - `c3_type` (línea 30): define el tipo asociado al modelo siendo el nombre de la subclase su valor por defecto.
  - `c3_module` (línea 31): define el módulo asociado al modelo siendo su valor por defecto el módulo «peat».
  - `c3_include` (línea 32): define si se hace una carga adelantada de información de las asociaciones del modelo. El valor por defecto es que no se haga ninguna carga adelantada.

  Finalmente en la línea 33 se configura que el modelo por defecto no hace uso de un repositorio de memoria.

- `c3_cached` (líneas 36 a 42): este método permite indicar que el modelo hace uso de un repositorio de memoria y las opciones de configuración para éste. Para activar el repositorio de memoria es necesario tener una instancia definida en `Bezel.cache`, si esto ocurre entonces se hace un `include` del módulo `Bezel::CacheBase` (línea 38) que agrega el uso y mantenimiento del repositorio de memoria a las acciones básicas y de búsqueda del modelo.

Métodos de instancia

Los métodos de clase que se definen en `Bezel::Base` son para inicializar un nuevo modelo, obtener la información sobre un atributo del modelo y las asociaciones con otros modelos

Cabe recordar que el servicio web C3 hace uso del formato JSON en sus respuestas a las peticiones al servicio. El formato JSON realiza una serialización de objetos, arreglos, números, cadenas, booleanos, y `null` con base en la sintaxis de Javascript[4].

En JSON los objetos son colecciones de llave-valor en donde la llave siempre es una cadena y el valor es alguno de los tipos básicos permitidos en JSON, esta estructura permite una biyección directa entre el formato JSON y tablas *hash* en Ruby.

```
1 json = '{"obj":{"id":"1340940799","name":"bezeltest"}}'
2 hash = JSON.parse(json)
3 # {"obj"=>{"id"=>"1340940799", "name"=>"bezeltest"}
4 hash["obj"]["id"]
5 # "1340940799"
```

En el código anterior tenemos una cadena con un objeto definido según el formato JSON en la línea 1. Haciendo uso de la biblioteca JSON en la línea 2 se parsea esta cadena y nos regresa una tabla *hash* la cual contiene la información definida en la cadena JSON original.

En Bezel se tiene que los atributos de los modelos obtenidos por medio del servicio web C3 son almacenados en una tabla *hash* interna, permitiendo el acceso a la información de una forma más orientada a objetos por medio de atributos y métodos de acceso.

```
1  module Bezel
2    class Base
3
4      # ...
5
6      attr_accessor :errors
7
8      def initialize(attributes = Hashie::Mash.new,
9                     persisted = false)
10       # ...
11       load(attributes)
12     end
13
14     def load(attributes)
```

[4]La principal divergencia es en el manejo de la codificación de las cadenas.

```
15      # ...
16    end
17
18    def method_missing(name, *args, &block)
19      if args.empty?
20        original_name = name.to_s
21        return @attributes[original_name] if @attributes.key?(
    original_name)
22        camelized_name = name.to_s.camelize(:lower)
23        return @attributes[camelized_name] if @attributes.key?(
    camelized_name)
24        return nil
25      end
26      super
27    end
28
29    def id
30      @attributes["id"]
31    end
32
33    # ...
34
35  end
36 end
```

En la mayoría de los lenguajes de programación que soportan el paradigma orientado a objetos se lanza una excepción cuando se llama un método que no está implementado por la clase del objeto o en la jerarquía de clases del objeto. En el caso de Ruby se lanza la excepción `NoMethodError` para indicar que no se encontró la implementación del método invocado sobre un objeto, sin embargo Ruby permite al programador manejar esta excepción por medio del método `method_missing` permitiendo que un objeto pueda responder a una conjunto de mensajes en forma dinámica.

El método `method_missing` es un método `callback` que recibe como argumentos:

- *name*: el nombre del método sin implementar en el objeto
- **args*: un arreglo con los argumentos originales con los que se llamó al método sin implementar.

Implementando el método `method_missing` se permite que los modelos definidos usando `Bezel::Base` puedan responder de forma dinámicamente a mensajes para acceder a los atributos del modelo.

Los métodos principales son:

- `initialize` (líneas 8-12): este método es el constructor por defecto en Ruby es invocado cuando se crea objeto nuevo. Los datos del modelo son obtenidos por medio del parámetro `attributes` que es una tabla *hash*
- `load` (línea 14): este método permite el cargar en un objeto ya existente nuevos valores de sus atributos.

- `method_missing` (líneas 18-27): como se explicó anteriormente se hace uso de este *callback* para permitir el acceso de los atributos del modelo en forma dinámica sin tener que definir explícitamente métodos de acceso para cada atributo del modelo. En la línea 21 se trata de ver si el nombre del método es una llave en la tabla *hash* de los atributos, si es así se regresa el valor contenido en la tabla. En la línea 23 se hace el mismo tipo de búsqueda pero se transforma el nombre al estilo de tipografía *lowerCamelCase*. Si ninguna llave corresponde se regresa `nil`.

- `id` (líneas 29-31): como se puede observar el método regresa el valor asociado a la llave `id` en la tabla de atributos del modelo este método de acceso no es necesario dada la implementación utilizada en `method_missing` pero esta solución tiene sus costos en rendimiento puesto que la invocación a un método es más eficiente que pasar por toda la búsqueda de un método inexistente hasta llegar a la invocación del método `method_missing`.

  En la implementación total de `method_missing` se hace uso de `define_method` para crear un método de acceso de forma dinámica para hacer mas eficiente el acceso posterior al atributo. Así para varios atributos básicos como `id` se define una implementación estática.

`Bezel::Associations`

En el módulo `Bezel::Associations` se definen métodos para definir asociaciones entre modelos basados en `Bezel::Base`. Este módulo es agregado por medio `extend` por lo que son métodos de clase

```
1  class EnergyConservationOption < Bezel::Base
2    c3_cached
3    set_c3_include "[this, {productCategory: [id]}, {faqs: [id]}]
       "
4
5    has_one :productCategory, ProductCategory
6    has_many :faqs, RecommendationFAQ
7  end
8
9  eco = EnergyConservationOption.find(10)
10 eco.productCategory
11 eco.faqs
```

En el código anterior se tiene la definición del modelo `EnergyConservationOption` el cual representa una característica del perfil de un edificio que está ligada a una recomendación. Para este modelo se definen dos asociaciones por medio del uso de los métodos `has_one` y `has_many`. La primera por medio de `has_one` (línea 5) se define una relación uno a uno con el modelo `ProductCategory` que representa una categoría de un catálogo de productos. Y la segunda con el método `has_many` (línea 6) se define una relación uno a muchos con el modelo `RecommendationFAQ`.

En las líneas 10 y 11 se tiene el uso de los métodos `productCategory` y `faqs`, los cuales regresan los modelos asociados al modelo, éstos son creados dinámicamente por

los métodos has_one y has_many.

```
1  module Bezel::Associations
2    def associations
3      @associations ||= {}
4    end
5
6    def has_many(field_name, model_name)
7      model = get_model_class(model_name)
8      define_assoc_method(field_name, model, false)
9    end
10
11   def has_one(field_name, model_name)
12     model = get_model_class(model_name)
13     define_assoc_method(field_name, model)
14   end
15
16   private
17
18   def get_model_class(model_name)
19     model = model_name.to_s.constantize
20     associations[field_name] = model
21     model
22   end
23
24   def define_assoc_method(field_name, model, one = true)
25     define_method(field_name) do
26       value = @attributes[field_name.to_s] || @attributes[
       field_name.to_sym]
27       if value
28         if one
29           model.find(value["id"])
30         else
31           value.map {|v| model.find(v["id"]) }
32         end
33       else
34         if id = @attributes["#{field_name}_id"]
35           res = model.find(id)
36         else
37           res = model.find(:all,
38             filter: "#{c3_type.to_s.camelize(:lower)}.id == '#{
       @attributes["id"]}'")
39           one ? res[0] : res
40         end
41       end
42     end
43   end
44 end
```

En la implementación de ambos métodos se hace uso de dos métodos auxiliares get_model_class y define_assoc_method. En las líneas 7 y 11 se hace uso de

`get_model_class` para obtener la clase del modelo con la que se está haciendo la asociación. Posteriormente en las líneas 8 y 13 se hace uso del método `define_assoc_method` para definir dinámicamente un método de instancia que realiza la recuperación y búsqueda de los modelos asociados.

En las líneas 18-22 se tiene la implementación del método `get_model_class` donde se hace uso del método `constantize` que realiza la recuperación de una constante a partir de una cadena dada, en este caso para recuperar de una cadena una clase.

En las líneas 24-43 se tiene la implementación del método `define_assoc_method` el cual hace uso del método `define_method` para construir un método de instancia el cual se encarga de realizar la búsqueda de los objetos asociados al modelo.

`Bezel::Connections`

En el módulo `Bezel::Connections` se definen los métodos que permiten realizar las acciones básicas sobre el modelo. La implementación de este módulo sigue un patrón bastante común en el cual se definen métodos tanto de clase como de instancia en un mismo módulo.

```
1  module Bezel::Connections
2    def self.included(base)
3      base.extend(ClassMethods)
4    end
5
6    module ClassMethods
7      def all(params = {})
8        params[:include] ||= c3_include if c3_include
9        action = invoke(:fetch, spec: params)
10
11       if action.result["count"] == 0 || action.result["objs"].
       nil?
12         []
13       else
14         action.result["objs"].map {|o| new(o) }
15       end
16     end
17
18     def one(params = {})
19       # ...
20       action = invoke(:fetch, spec: params)
21       # ...
22     end
23
24     def upsert(attrs)
25       # ...
26       action = invoke(:upsert, obj: attrs)
27       # ...
28     end
29   end
30
```

```
31   def save
32     # ...
33     self.class.upsert(self.to_hash)
34     # ...
35   end
36
37   def update(update)
38     # ...
39     action = self.class.invoke(:upsert, obj: update, srcObj:
       old)
40     # ...
41   end
42
43   def destroy
44     self.class.invoke(:remove, obj: self.to_hash)
45   end
46 end
```

En la línea 2 se tiene la implementación del método `included` que es un *callback* que es invocado cuando el módulo `Connections` es utilizado en un `include`. En la línea 3 al incluirse el módulo en una clase o módulo `base` se agregan como métodos de clase los métodos definidos en el submódulo `ClassMethods`.

Los métodos de clase más importantes son:

- `all` (líneas 7-16): este método realiza una búsqueda dentro del modelo. En la línea 8 se toma en cuenta la configuración `c3_include` para indicar que modelos asociados se deben precargar en la búsqueda. En la línea 9 se realiza la petición usando la acción `fetch` la cual permite hacer búsquedas en el servicio web C3. En las líneas 11 a 15 se tiene la lógica necesaria para manejar el resultado de la petición del servicio web C3, cabe señalar que el resultado de la petición es un arreglo de tablas `hash`, cada tabla es usada para crear un nuevo modelo (línea 14) para regresar finalmente un arreglo de modelos como espera el programador.
- `one` (líneas 18-22): este método realiza una búsqueda la cual regresa solamente el primer objeto que cumpla el criterio de búsqueda. En la peticion al servicio web se hace uso de la acción `fetch` (línea 20).
- `upsert` (líneas 24-28): este método maneja tanto la creación como la actualización de modelos, espera como argumento una tabla *hash* con los atributos del modelo a crear o actualizar. La acción `upsert` es usada para realizar la petición al servicio web (línea 26).

Aunque el método de clase `upsert` permite al programador el crear o actualizar los modelos se definen métodos de instancia para facilitar cada caso de uso específico.

- `save`: este método de instancia hace uso de `upsert` y de la representación *hash* del modelo (`to_hash`) para guardar el modelo en el servicio web C3.
- `update`: este método de instancia realiza la actualización del modelo siendo más eficiente en tiempo de ejecución, ya que solo se actualizan los atributos que han sido cambiados.

- `destroy`: este método elimina el modelo por medio de la acción `remove` (línea 44).

`Bezel::CacheBase`

El módulo `Bezel::CacheBase` mejora los métodos definidos en `Connections` permitiendo que las acciones básicas y de búsqueda hagan uso de un repositorio de memoria (`Bezel::Cache`).

El módulo sigue el mismo patrón de agrupación de los métodos de clase en el submódulo `ClassMethods` así como los métodos de instancia en el módulo principal, haciendo uso del método *callback* `included` para agregar los métodos de clase.

```
1  module Bezel::CacheBase
2    def self.included(base)
3      base.extend(ClassMethods)
4    end
5
6    module ClassMethods
7      attr_reader :memo
8
9      if Bezel.cache && !Bezel.cache.class == Bezel::Cache
10       Bezel.cache = Bezel::Cache.new
11     end
12
13     def prime
14       all
15     end
16
17     def cache_read(key)
18       Bezel.cache.read(key_for(key), @cache_type)
19     end
20
21     def cache_write(key, value)
22       Bezel.cache.write(key_for(key), value, @cache_type,
       @cache_opts)
23     end
24
25     def cache_delete(spec_or_id)
26       Bezel.cache.delete(key_for(spec_or_id), @cache_type)
27     end
28
29     def all(params = {})
30       params[:include] = c3_include if c3_include
31       tmp = cache_read(params)
32
33       if tmp
34         tmp.map {|id| one(id)}
35       else
36         records = super
37         cache_write(params, records.map {|r| r.id})
```

```
38          records.each do |record|
39            cache_write(record.id, record.to_hash(
    with_associations: false))
40          end
41          records
42        end
43      end
44
45      def one(params = {})
46        # ...
47      end
48
49      def key_for(id_or_key, scoped = false)
50        if @cache_type == :shared
51          prefix = "#{Bezel.tenant}-#{Bezel.tag}-#{model_name}"
52        else
53          prefix = "#{Bezel.tenant}-#{Bezel.tag}-#{model_name}-#{
    Bezel.client.auth_token}"
54        end
55
56        if id_or_key.respond_to?(:to_hash) || scoped
57          id_or_key[:include] = c3_include if c3_include
58          "#{prefix}|all(#{id_or_key})"
59        else
60          "#{prefix}|#{id_or_key}"
61        end
62      end
63    end
64
65    def save
66      # ...
67    end
68
69    def update(attrs)
70      result = super
71      cache_delete(self.id)
72
73      result
74    end
75
76    def destroy
77      # ...
78    end
79  end
```

En las líneas 9-11 se inicializa el repositorio de memoria con una instancia de `Bezel::Cache` si es necesario.

En las líneas 13-27 se agregan métodos auxiliares de clase para operaciones de lectura, escritura y borrado de modelos en el repositorio de memoria.

En las líneas 29-43 se tiene la implementación del método `all` haciendo uso del método `cache_read` (línea 31) para obtener los modelos por medio del repositorio de memoria, si los modelos no se encuentren en el repositorio se llama a `super` (línea 36) que ejecuta la implementación del método `all` definido en `Bezel::Connections` el cual realiza una petición al servicio web C3. El resultado de la petición es almacenado en el repositorio de memoria por medio del método `cache_write` (líneas 37 y 39).

Los métodos `one`, `save`, `update` y `destroy` siguen el mismo patrón de implementación visto en el método `all`, es decir, se realiza alguna operación en el repositorio de memoria y opcionalmente se delega la ejecución de la petición a la implementación del método definida en `Bezel::Connections` por medio de `super`.

El método auxiliar `key_for` es usado para construir las llaves usadas para almacenar los modelos en el repositorio de memoria, la generación de la llave depende del tipo de repositorio seleccionado para el modelo.

3.4.6. `Bezel::Cache`

La clase `Cache` representa un repositorio de memoria y las operaciones para manipularlo directamente. Esta clase es usada principalmente por `Bezel::CacheBase` para almacenar las respuestas a peticiones realizadas al servicio web C3.

Se tienen dos tipos de repositorio de memoria:

- *shared*: en este repositorio se almacenan valores que son utilizados globalmente en el sistema.
- *local*: en este repositorio se almacenan valores específicos a un usuario. Se usa el elemento de autenticación (*auth_token*) del usuario para generar las llaves para el repositorio de memoria local.

```
1  class Bezel::Cache
2    def initialize(config = {local: true, shared: true})
3      @caches = {}
4      [:local, :shared].each do |t|
5        @caches[t] = if (cache_config = config[t]) &&
       cache_config.respond_to?(:"[]")
6          if /Cassandra/ =~ cache_config[:class]
7            require 'bezel/moneta/cassandra'
8          end
9          cache_config[:class].constantize.new(cache_config[:
       options] || {})
10       else
11         ::Moneta::Adapters::Memory.new
12       end
13     end
14   end
```

```
15
16    attr_reader :caches
17
18    def read(key, type = :local)
19      @caches[type][key]
20    end
21
22    def write(key, value, type = :local, opts = {})
23      @caches[type].store(key, value, opts)
24    end
25
26    def remove_id(key, id, type = :local)
27      # ...
28    end
29
30    def delete(spec_or_id, type = :local)
31      # ...
32    end
33
34    def update(id, new_value, type = :local, opts = {})
35      # ...
36    end
37  end
```

En las líneas 2-14 se tiene el constructor para la clase el cual dependiendo de los parámetros de configuración hace uso del adaptador corrector para inicializar el repositorio de memoria. Por medio de la biblioteca `Moneta` se logra definir un API uniforme para manipular un repositorio de memoria teniendo como `backend` una gran cantidad de repositorios de tipo llave-valor[5]. En las líneas 18-20 se tiene la implementación del método `read` que permite recuperar un valor del repositorio de memoria dada una llave y el tipo de repositorio. En la líneas 22-28 se tiene la implementación del método `write` que permite asociar una llave con un valor en el repositorio de memoria.

Esta clase se implemento para abstraer el uso de un repositorio de memoria y no acoplar las operaciones de repositorio de memoria a una sola biblioteca como `Moneta` o a un repositorio llave-valor en particular como `Memcached` dando flexibilidad de cambiar de biblioteca o repositorio.

En este capítulo se analizaron las diferentes decisiones de arquitectura y diseño que se tomaron para satisfacer los requerimientos del sistema `PEAT`. Se vio con detalle la implementación de los casos de uso del sistema poniendo énfasis en la implementación de la interfaz de usuario y la biblioteca `Bezel`. También se mostró el efecto de las pruebas con usuarios durante la implementación de la interfaz, donde se observó la necesidad de simplificar principalmente la creación del perfil de edificio. En la implementación de `Bezel` se mostraron los beneficios de la metaprogramación para la rápida implementación de modelos basados en recursos provenientes de servicios web de tipo RESTful.

[5] Moneta tiene adaptadores para Memcached, Redis, Cassandra, Berkeley DB, MongoDB, etcétera.

Capítulo 4

Despliegue del sistema

El despliegue de un sistema es una etapa de creciente importancia dado que los sistemas actuales van requiriendo más servicios y subsistemas de apoyo para su funcionamiento. En la actualidad para garantizar un servicio estable para un gran número de usuarios es necesario contar con repositorios de memoria, balanceadores de carga, servidores de cola, etcétera. Esta creciente complejidad incrementa los riesgos para lograr un despliegue a producción exitoso.

En este contexto se encuentra PEAT puesto que uno de sus requerimientos principales es dar soporte a por lo menos mil usuarios concurrentes, en este capítulo se describe la arquitectura del ambiente de producción, el proceso de despliegue continuo para el despliegue del sistema PEAT en los ambientes de prueba y producción, así como las optimizaciones realizadas tanto al sistema como a la arquitectura de los ambientes para permitir un mayor rendimiento posible.

4.1. Arquitectura del ambiente de producción

Para el despliegue de los ambientes requeridos para PEAT se hizo uso de *Amazon Web Services* (AWS) en particular del servicio *Amazon Elastic Compute Cloud* (EC2) que permite la renta de servidores para correr aplicaciones.

El uso de EC2 presentaba las siguientes ventajas:

- Permite un gran márgen de maniobra en la configuración de los servidores puesto que permite elegir entre una gran variedad de sistema operativos, capacidad de cómputo, etcétera.
- Permite la creación y configuración automatizada de los servidores permitiendo que el sistema pueda escalar rápidamente según las necesidades de cómputo.
- Una comunicación más rápida y segura con el backend dado que el servidor C3 se ejecuta también bajo servidores EC2.

Se hizo uso de los siguientes tipos de nodos:

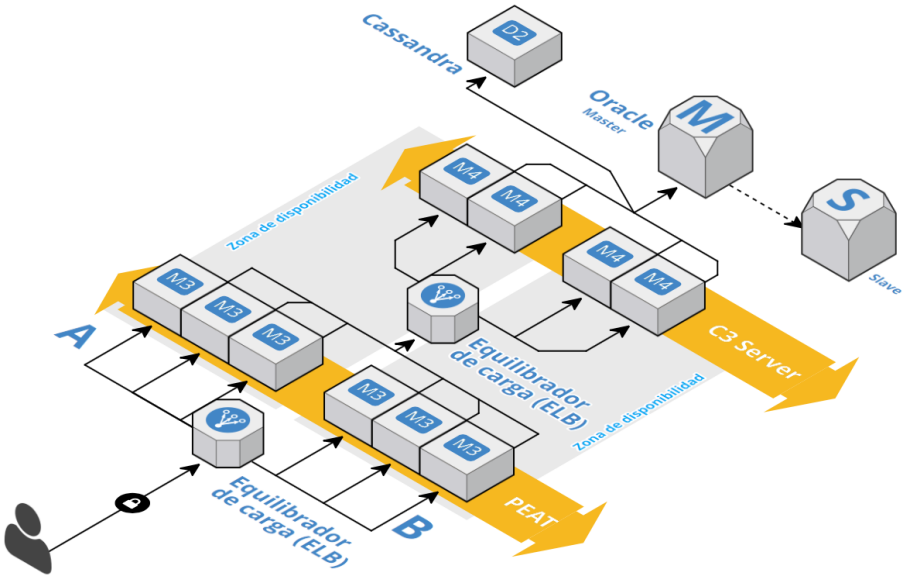

Figura 4.1: Arquitectura ambiente de producción.

- M3: estos nodos cuentan con 2 CPU virtuales, 3.75 GB de memoria y 410 GB de almacenamiento HDD.
- M4: estos nodos cuentan con 4 CPU virtuales, 15 GB de memoria y 850 GB de almacenamiento HDD.

Cada CPU virtual proporciona la capacidad de cómputo equivalente a un CPU Xeon de 1.2 GHz del 2007.

El ambiente de producción tiene los siguientes componentes (Ver Figura 4.1):

- Balanceadores de carga: AWS tiene el servicio *Elastic Load Balancing* (ELB) el cual permite obtener niveles altos de tolerancia al distribuir el tráfico de forma automática entre varias instancias EC2 las cuales se encuentran en varias zonas de disponibilidad[1] (Ver Figura 4.1 ELB).
- Nodos EC2: son servidores con capacidades de cómputo que pueden ser activadas en cuestión de minutos. Para el sistema `PEAT` se usan seis instancias del tipo M3. Para el servidor C3 se usan cuatro instancias de tipo M4 (Ver Figura 4.1 M3 y M4).
- Oracle BD: base de datos usada para almacenar la información usada por el servidor C3 es administrada por una base de datos Oracle Database 10g (Ver Figura

[1]Zonas que se encuentran en ubicaciones físicas distintas que cuentan con su propia infraestructura, independiente y físicamente distinta.

4.1 Oracle). Se tienen cuatro nodos por medio del servicio *Amazon RDS for Oracle Database* los cuales cuentan con 16 CPU virtuales y 64 GB de memoria.

- Apache Cassandra: un *cluster* de diez nodos de tipo M4 ejecutando Apache Cassandra el cual tiene las siguiente funciones (Ver Figura 4.1 Cassandra):
  - Recopilar y analizar grandes volúmenes de datos en secuencia, como el historial de consumo de energía obtenido por los medidores inteligentes[2].
  - Como capa de repositorio de memoria del servidor C3.

Se pude observar en la figura 4.1 dos zonas de disponibilidad esto permite tener una mayor redundancia al no tener un único punto de error.

Se tienen dos balanceadores de carga el primero dirige las peticiones de los usuarios a una instancia EC2 que corre instancias del sistema PEAT, el segundo balanceador de carga es usado para distribuir la carga de peticiones al servidor C3 por parte de todas las instancias PEAT.

4.1.1. Phusion Passenger y Nginx

Para el funcionamiento de PEAT es necesario tener tanto un servidor web como un servidor de aplicación. Para PEAT se eligió utilizar Phusion Passenger como servidor de aplicación por las siguientes razones:

- Gran soporte para sistemas implementados con Ruby.
- Crea y apaga instancias del sistema según la demanda, lo que permite tener buen rendimiento cuando se tiene mucha demanda pero conservando recursos cuando sea posible.
- Reduce el uso de memoria para el sistema, por medio de técnicas como la precarga de aplicación y usando la gestión de memoria de copia en escritura (*copy-on-write*, COW).
- Hace uso de todos los núcleos de CPU disponibles para el sistema.

Phusion Passenger permite la integración con los servidores web Apache o Nginx, haciendo que estos se encarguen tanto del manejo de archivos estáticos [3] como de distribuir la carga de peticiones entrantes.

Se eligió integrar Nginx como servidor web por dos razones (1) su configuración es mucho más simple que en Apache y (2) permite fácilmente agregar encabezados a las peticiones entrantes sin afectar el rendimiento del sistema lo que permitió un análisis más profundo de las peticiones procesadas por el sistema PEAT.

De esta forma tenemos que en cada instancia M3 se tiene los siguientes componentes (Ver Figura 4.2):

[2] Al escribir datos en Cassandra, éstos se ordenan y se guardan en forma secuencial en disco, por lo que para recuperar los datos solo se necesita de la clave de fila y un rango, logrando un patrón de acceso rápido y eficiente [5]

[3] Imágenes, hojas de estilo, código Javascript, etcétera

- Nginx: proporciona los archivos estáticos del sistema y es también el balanceador de carga de las instancias (Ver Figura 4.2 Nginx).
- *Preloader*: es un proceso que carga todo el sistema PEAT, este proceso es copiado para obtener las instancias PEAT que manejan las peticiones (Ver Figura 4.2 Nginx Preloader).
- Instancias PEAT: son instancias del sistema PEAT obtenidos por medio de *Preloader* el número mínimo de instancias se fijo en diez teniéndose la capacidad de crear hasta veinte instancias en forma automática.

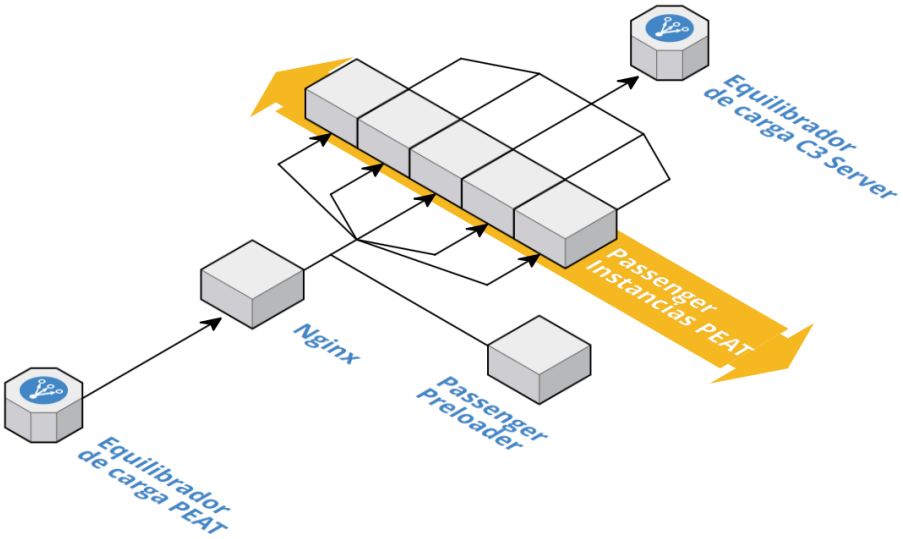

Figura 4.2: Arquitectura ambiente de producción.

Como se ve en la Figura 4.2 el servidor Nginx recibe una petición por parte del balanceador de carga para PEAT, dependiendo de la carga en las instancias PEAT la petición se manda a una instancia PEAT libre o es puesta en una cola de espera. Passenger detecta si la carga va aumentando y determina automáticamente si es necesario crear nuevas instancias del sistema PEAT por medio del proceso `Preloader`.

Aunque en la Figura 4.2 se tienen cinco instancias PEAT ejecutándose en producción se tienen por lo menos diez instancias ejecutándose en todo momento con la posibilidad de crearse hasta 20 instancias al tenerse más tráfico.

Mediante varias pruebas de carga se determinó que el sistema debía manejar picos de demanda de hasta 30,000 peticiones por minuto para dar soporte al menos 1,000 usuarios concurrentes. Cada instancia PEAT es capaz de procesar hasta 300 peticiones por minuto por lo que cada servidor M3 procesa entre 3,000 a 6,000 peticiones por minuto. Para el ambiente de producción se determinó usar seis servidores M3 por lo que la capacidad base es de 18,000 peticiones por minuto pero con capacidad de crecer con la demanda hasta 36,000 peticiones por minuto.

Los ambientes de prueba tenían la misma estructura solo cambiando el número de servidores según las necesidades y pruebas a realizar.

4.1.2. Inicialización e invalidación del repositorio de memoria

Uno de los principales cuellos de botella encontrados durante el desarrollo del sistema PEAT fue la gran cantidad de peticiones realizadas al servicio web. Para reducir el número de peticiones se determinó que un gran número de éstas se podían evitar teniendo un repositorio de memoria de los modelos que solo pueden ser modificados por el administrador de sistema PEAT y que además son actualizados cada seis meses.

Los modelos que se pueden considerar que contienen información estática son:

- *BuildingQuestion*: este modelo representa una pregunta sobre el perfil de un edifico, se tienen 45 preguntas por tipo de edificio, en total se tienen 500 preguntas.
- *Recommendation*: este modelo representa una recomendación con todos su información de costo, beneficios, etcétera. Se inicio con un total de cien recomendaciones.
- *ValidAnswer*: las respuestas válidas para una pregunta, dado que la mayoría de las preguntas tiene cuatro posibles respuestas se tiene un estimado de dos mil respuestas en el sistema.

Se determinó que la mejor solución fue que al iniciar el sistema PEAT se precargara el repositorio de memoria con la información de los modelos descritos anteriormente por medio del servicio web C3. De esta forma se redujo el número de peticiones de forma substancial pero teniendo el inconveniente de un incremento en el tiempo de inicialización de una instancia del sistema PEAT.

Para minimizar el impacto de un mayor tiempo de inicialización se hizo uso de las características de precarga y de gestión de memoria de copia en escritura (*copy-on-write*, COW) proporcionados por Phusion Passenger. Al iniciar Phusion Passenger se inicia el proceso `Preloader` que realiza la inicialización del sistema PEAT y la precarga de modelos en el repositorio de memoria.

En la Figura 4.3 se puede ver que por medio de *Preloader* se comparte en memoria el código del marco de trabajo Rails, el código del sistema PEAT y el repositorio de memoria de los modelos descritos anteriormente. Cada instancia solo usa memoria para sus propios objetos utilizados para manejar las peticiones entrantes.

De esta manera al iniciar una nueva instancia del sistema se realiza una copia del proceso *Preloader* el cual contiene los modelos precargados en el repositorio de memoria por lo que la inicialización de nuevas instancias es casi instantánea.

Para la invalidación del repositorio de memoria de los modelos se tenia una acción para el administrador del sistema que permite limpiar el repositorio de memoria y actualizar los modelos por medio del servicio web C3.

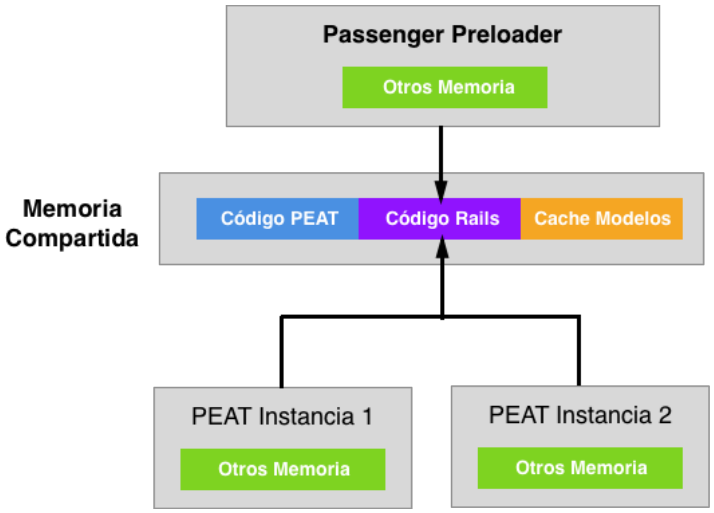

Figura 4.3: Diagrama de la memoria compartida entre *Preloader* y las instancias PEAT.

4.2. Despliegue continuo

El despliegue continuo (*Continuous Delivery*, CD) es una buena práctica de software en la cual se implementa software de tal forma de que éste pueda ser desplegado a producción en cualquier momento[10]. Su objetivo es crear, probar y liberar software más rápido y con mayor frecuencia.

Se considera que un sistema implementa el despliegue continuo cuando [10]:

- El sistema puede ser desplegado a producción en cualquier momento del ciclo de desarrollo.
- Se tiene una rápida retroalimentación sobre la capacidad del sistema para ser desplegado en producción al realizarse cambios en el sistema.
- Se puede desplegar de forma sencilla cualquier versión del software para cualquier entorno.

Los principales beneficios de esta práctica son:

- Reducción del riesgo de despliegue: dado que se realizan despliegues a producción de forma recurrente los cambios contenidos en cada despliegue son menores por lo que hay menores posibilidades de errores y si un error se presenta es más fácil de localizar y arreglar.
- Retroalimentación del usuario: uno de los mayores riesgos en la ingeniería del software es el desarrollar un sistema que no es útil para el cliente o usuario. Por lo tanto obtener retroalimentación sobre el desarrollo del sistema de forma rápida y frecuente permite evaluar que tan útil es el sistema para el usuario.

En PEAT los riesgos asociados al despliegue de producción exitoso eran mayores a lo habitual, dada la complejidad del sistema y la cantidad de requerimientos. Además PG&E requería el despliegue del sistema en ambientes de pruebas previo a un despliegue del sistema en el ambiente de producción. La implementación de un proceso de despliegue continuo para PEAT fue de vital importancia para cumplir con las necesidades del cliente.

Para PG&E era necesario que se tuvieran los siguientes ambientes:

- *peattest*: este ambiente es usado para hacer pruebas con usuarios seleccionados por PG&E para obtener retroalimentación sobre el sistema.
- *peatqa*: este ambiente es usado por PG&E para realizar control de calidad y de rendimiento.
- *peatprod*: este ambiente es el ambiente final de producción.

Para uso interno en C3 Energy se tenía además el ambiente *peatstage*, el cual era usado para realizar pruebas de rendimiento y de control de calidad.

En total se contaba con cuatro ambientes, los cuales debían ser lo más parecidos en su arquitectura y configuración a la del ambiente de producción para que así las pruebas de calidad y de rendimiento regresaran resultados cercanos al comportamiento del sistema en el ambiente de producción. Dados estos requerimientos la implementación del proceso de despliegue continuo para el sistema PEAT era la mejor forma para manejar el despliegue del sistema en esta diversidad de ambientes.

4.2.1. Proceso de despliegue

En el despliegue continuo se tiene al proceso de despliegue (*deployment pipeline*) como concepto central, el cual, en esencia es la implementación automatizada de los procesos de configuración, construcción, prueba y despliegue de un sistema [13].

Este proceso tiene las siguientes ventajas:

- Visibilidad: todas las etapas del sistema de despliegue son visibles para todos los miembros del equipo.
- Retroalimentación: los miembros del equipo obtiene información sobre los problemas en el momento en que ocurren de modo que son capaces de dar solución a éstos rápidamente.
- Despliegue: por medio de un proceso totalmente automatizado se puede desplegar y liberar cualquier versión del sistema en cualquier ambiente y en cualquier momento.

El proceso de despliegue se conforma de una serie de etapas de validación por las que el sistema debe pasar para llegar al ambiente de producción. Para el sistema PEAT las etapas de validación son las siguientes (Ver Figura 4.4):

- Control de versiones: el equipo de desarrollo ingresa los cambios realizados al sistema por medio del control de versiones Git, esto inicia el proceso de despliegue.

- Pruebas de unidad: al detectarse un cambio en el repositorio se ejecutan las pruebas de unidad del sistema.

  Se usa la biblioteca `RSpec` para implementar las pruebas de unidad para el código en Ruby, mientras que para el código en Javascript se hace uso de la biblioteca `Jasmine` para las pruebas de unidad.

  El equipo de desarrollo recibe retroalimentación del resultado de las pruebas, si el resultado es positivo se pasa a la siguiente etapa. En la primera secuencia en la Figura 4.4 se tiene el caso cuando las pruebas de unidad son negativas y se detiene el proceso de despliegue.

- Pruebas de aceptación: se hace uso de la biblioteca *Cucumber* para implementar las pruebas de aceptación, de la misma forma que la etapa de pruebas de unidad si se obtiene un resultado positivo se continúa a la siguiente etapa en caso contrario se detiene el proceso de despliegue.

- Ambientes de prueba: se hace el despliegue del sistema en los ambientes de prueba, es decir, a los ambientes *peatstage*, *peattest* y *peatqa*. En esta etapa se realizan pruebas de rendimiento y de calidad de forma manual y automatizada.

- Producción: se hace un despliegue a producción cuando la versión en los sistemas de prueba han sido revisado de forma satisfactoria.

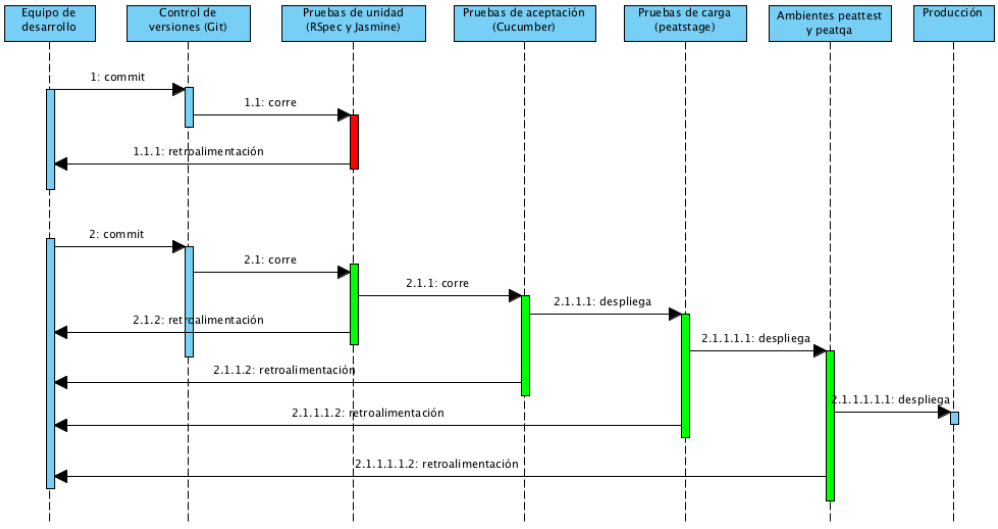

Figura 4.4: Proceso de despliegue para `PEAT`.

Las primeras etapas del proceso se desarrollan en forma completamente automatizada hasta llegar a la etapa de despliegue a los ambientes de pruebas, en esta etapa al principio se realizaban pruebas de calidad y rendimiento de forma manual, estas actividades se fueron automatizando una por una haciendo uso de herramientas como `SauceLabs` y `CloudTest`. La ejecución para el despliegue al ambiente de producción se realiza de forma manual.

Herramientas

El proceso de despliegue se hace uso de un gran número de herramientas, las más importantes son:

- `AWS`: *Amazon Web Services* (AWS), es un conjunto de servicios de cómputo en la nube que permite la implementación de sistemas escalables.
- `Jenkins`: es un servidor de despliegue continuo, que permite definir las etapas de un proceso de despliegue y las acciones que se realizan según el resultado de la etapa.
- `Capistrano`: es una herramienta que facilita la creación de tareas para el despliegue de sistemas en servidores remotos.
- `CloudTest`: es una herramienta para la ejecución de pruebas de carga y rendimiento para aplicaciones móviles y web.
- `SauceLabs`: es una herramienta que permite ejecutar las pruebas de un sistema en mas de 500 combinaciones de navegadores, sistema operativos y dispositivos.

4.2.2. Configuración y despliegue automatizado de ambientes

En la Sección 4.1 se hablo sobre la arquitectura del ambiente de producción para `PEAT` en donde se indicó el uso de servidores EC2 proporcionados por *Amazon Web Services* (AWS).

Capistrano

Para configurar los servidores EC2 para `PEAT` se usa *Capistrano* que es una herramienta para definir y ejecutar tareas en múltiples servidores remotos por medio de SSH[4]. Esta herramienta define un DSL basado en Ruby para la definición de tareas de despliegue y mantenimiento.

Capistrano contiene tareas predefinidas para el despliegue de sistemás basados en el marco de trabajo Rails, sin embargo para el sistema `PEAT` se implementaron tareas adicionales para realizar un despliegue exitoso.

[4]Es un protocolo que permite acceder a servidores remotos de forma segura a través de una red [1].

Configuración de ambientes

```
1  set :config_defaults, {
2    appdirname: "peat",
3    c3server: "http://c3server.com:8080",
4    nginx_port: 8888,
5    rails_env: "production",
6    tenant: "peat",
7    tag: "prod",
8    # ...
9  }
10
11 set :stages, %w{peatstage peattest peatqa peatprod}
12 set :application, "peat"
13
14 set :scm, :git
15 set :repository,  "git@server.com:c3energy/peat.git"
16 set :branch, "#{ENV['branch'] || 'release/v1.0'}"
17
18 # Tareas para inicializar el ambiente
19 after "deploy:setup", "deploy:peat_setup"
20 before 'deploy:setup', 'rvm:install_rvm'
21 before 'deploy:setup', 'rvm:install_ruby'
22
23 # Se reinicia el sistema después de actualizar su configuración
24 after "peat:config_sync", "deploy:restart"
```

El código anterior muestra la configuración base de variables para el despliegue del sistema PEAT usando el DSL definido por Capistrano. En las líneas 1-9 se definen las variables de configuración para el sistema PEAT y el servidor Nginx, los valores por defecto están definidos para un ambiente en producción, lo cual facilita conocer las diferencias entre el ambiente de producción y los ambientes de pruebas.

En la línea 11 se definen los ambientes soportados, teniendo los cuatro ambientes descritos anteriormente, cabe señalar que cada ambiente tiene su propio archivo de configuración en donde se indican los cambios en variables y/o tareas con respecto a la configuración base.

En las líneas 14-16 se indica el sistema de versiones, la dirección del repositorio y la rama[5] a usar para obtener el código fuente del sistema. En PEAT el sistema de versiones es Git, donde la rama por defecto es «release/v1.0» que es la rama estable del sistema que se usa para producción, por medio de la variable de ambiente BRANCH es posible hacer el despliegue de cualquier versión del sistema.

En las líneas 18-24 se tiene el uso de las declaraciones `after` y `before` que permiten el establecer que una tarea sea realizada antes o después de una segunda tarea. Cabe señalar que las tareas `deploy:setup` y `deploy:restart` son tareas definidas por Capistrano, la primera es una tarea para la configuración inicial de un ambiente[6] y la segunda tarea es realizada al reiniciar el sistema.

[5] Es una versión etiquetada del código fuente en el control de versiones Git.

[6] Esta tarea es solo ejecutada una única vez para preparar un nuevo ambiente.

```
1  namespace :deploy do
2    desc "Crear archivos de configuración para upstart"
3    task :upstart do
4      run "cd #{current_path} && rvmsudo bundle exec foreman
       export upstart /etc/init -a #{application} -u nginx -t ./
       foreman"
5    end
6    desc "Iniciar sistema PEAT"
7    task :start do
8      sudo "/sbin/start #{application}"
9    end
10   desc "Parar sistema PEAT"
11   task :stop do
12     sudo "/sbin/stop #{application}"
13   end
14   desc "Transferir archivos de configuración para PEAT y Nginx"
15   task :peat_setup do
16     transfer_config_files(server_env)
17     run "cp #{shared_path}/nginx.conf /opt/nginx/conf/nginx.
       conf"
18   end
19 end
20
21 namespace :peat do
22   desc "Transferir archivos de configuración para PEAT y Nginx
       y el sistema se reinicia."
23   task :config_sync do
24     transfer_config_files(server_env)
25     run "cp #{shared_path}/nginx.conf /opt/nginx/conf/nginx.
       conf"
26   end
27 end
28
29 namespace :generate do
30   task :config do
31     ConfigCreator.new(config_defaults).write
32   end
33   task :jenkins do
34     config_defaults[:rails_env] = "test"
35     config_defaults[:c3server] = "https://peatstage-admin.
       c3server.com"
36     config_defaults[:tag] = "cucumber"
37     ConfigCreator.new(config_defaults).write
38   end
39 end
```

En el código anterior se tiene las tareas más importantes para el despliegue del sistema. En las líneas 3-5 se define la tarea `upstart` la cual por medio del comando `foreman` genera la configuración para agregar al sistema `PEAT` como un servicio al

sistema Upstart[7] usado en el sistema operativo Ubuntu 12.04 LTS que es el sistema operativo seleccionado para los servidores EC2.

En las líneas 6-13 se definen las tareas para iniciar y parar el sistema PEAT, se hace uso de los comandos `start` y `stop`, así como de la configuración generada para el sistema Upstart en la tarea anterior para iniciar o parar fácilmente el sistema.

En las líneas 15-19 se define la tarea `peat_setup` la cual genera los archivos de configuración para PEAT (`peat.yml`) y Nginx (`nginx.conf`) y posteriormente los transfiere a los servidores del ambiente, esta tarea se ejecuta solamente cuando se inicia un nuevo ambiente.

En las líneas 21-27 se define la tarea `config_sync` la cual realiza las tareas de generar los archivos de configuración, transferirlos y ejecuta el reinicio del sistema para que éste cargue los cambios de configuración.

En las líneas 30-35 se define la tarea `config` la cual genera los archivos de configuración para PEAT (`peat.yml`) y Nginx (`nginx.conf`) por medio de `ConfigCreator`.

```
1  set :hostname, 'dev-peatstage-ruby-01.c3-e.com'
2  server "#{hostname}", :app, :web, :primary => true
3
4  set :branch, "#{ENV['branch'] || 'master'}"
5  set :application, "pge"
6  set :deploy_to, "/home/nginx/rails_app/peat-pge"
7
8  set :appdirname, "peat-pge"
9  set :c3server, "http://peatstage.c3server.com:8080"
10 set :rails_env, "staging"
11
12 def override_defaults
13   config_defaults[:appdirname] = appdirname
14   config_defaults[:server_name] = server_name
15   config_defaults[:c3server] = c3server
16   config_defaults[:railsenv] = rails_env
17 end
18
19 def write_and_transfer_config_files
20   override_defaults
21   ConfigCreator.new(config_defaults).write
22   puts "Transfering config/peat.yml"
23   transfer(:up, "tmp/peat.yml", "#{shared_path}/peat.yml")
24   puts "Transfering nginx conf file"
25   transfer(:up, "tmp/nginx.conf", "#{shared_path}/nginx.conf")
26   run "cp #{shared_path}/nginx.conf /opt/nginx/conf/nginx.conf
       .#{application}"
27 end
28
29 namespace :deploy do
30   task :peat_setup do
31     write_and_transfer_config_files
```

[7] Es un demonio que maneja el inicio de tareas y servicios durante el arranque del servidor y la supervisión de éstos mientras el servidor está funcionando.

```
32   end
33 end
34
35 namespace :peat do
36   task :config_sync do
37     write_and_transfer_config_files
38   end
39 end
40
41 namespace :generate do
42   task :config do
43     override_defaults
44     ConfigCreator.new(config_defaults).write
45   end
46 end
```

Como se mencionó anteriormente cada ambiente tiene su propio archivo de configuración, el código anterior es la configuración para el ambiente `peatstage`.

En la línea 1-2 se define el servidor asociado al ambiente. En las líneas 3-10 se redefinen varias variables de configuración para el ambiente siendo uno de los principales cambios que la rama por defecto es «master» la cual es la rama principal de desarrollo en el sistema `PEAT`. En las líneas 29-46 se redefinen las tareas `peat_setup`, `config_sync` y `config` para que tomen en cuenta la configuración actualizada por medio del método `override_defaults`.

Despliegue automatizado de ambientes

La interacción con `Capistrano` es por medio del comando `cap` en la línea de comandos, así para hacer un despliegue al ambiente `peatprod` se ejecutaría el siguiente comando.

```
cap peatprod deploy
```

El comando `cap` toma dos argumentos el primero es algunos de los ambientes definidos en la variable `stages` y el segundo es el nombre de la tarea a realizar.

Como se puede ver por medio de `Capistrano` se puede realizar tanto la configuración inicial de los ambientes y el despliegue del sistema `PEAT`.

El uso de `Capistrano` para el despliegue tiene las siguientes ventajas:

- Rápidamente se puede agregar un nuevo ambiente agregando su nombre a la variable `stages` y agregando su archivo de configuración con los cambios deseados para este nuevo ambiente.
- La configuración definida con `Capistrano` es parte del código fuente del sistema y se encuentra en el sistema de versiones por lo que se tiene historia de los cambios realizados en la configuración de los ambientes.
- El sistema es auto contenido, es decir, el código para realizar el despliegue del sistema es parte del código fuente del sistema.

En el servidor *Jenkins* se implementó una tarea que al tenerse un resultado positivo de las pruebas del sistema se hace uso del comando `cap` para efectuar el despliegue del sistema en los ambientes de prueba.

4.2.3. Pruebas de unidad y aceptación

En el servidor *Jenkins* se definen las etapas del proceso de despliegue, siendo el inicio del proceso de despliegue cuando se detecta un cambio en el repositorio del sistema. Al detectarse este cambio se ejecuta la primera etapa que implica ejecutar las pruebas de unidad.

Pruebas de unidad

En la sección 2.3.2 se habló sobre la práctica del desarrollo guiado por comportamiento (*Behavior Driven Development*, BDD) la cual involucra escribir las pruebas antes de escribir el código fuente y la refactorización continúa en el código fuente.

En el desarrollo guiado por comportamiento las pruebas se enfocan en describir el comportamiento y la interacción de los módulos del sistema. Para el sistema `PEAT` se hace uso del marco de trabajo *RSpec* la cual define un Lenguaje de Dominio Específico (LDE) para implementar las pruebas unitarias.

RSpec da acceso al comando `rspec` que permite ejecutar y verificar que el sistema pasa todas las pruebas unitarias del sistema. En la primera etapa el servidor *Jenkins* obtiene el código fuente del sistema y ejecuta el comando `rspec` el cual ejecuta todas las pruebas unitarias del sistema.

```
1  require 'spec_helper'
2
3  describe QuestionsController do
4    before :each do
5      @building = mock_model(Building, id: "building_1",
6                             placed_at_account_id: 1,
7                             service_location_account_id: 1,
8                             recommendations_valid: true,
9                             locations: [])
10     Building.stub(:find) { @building }
11     @bpa = mock_model(BuildingProfileAnswer, id: "bpa_1",
       to_partial_path: "questions/text")
12     PEATEngine.stub(:next_question) { @bpa }
13   end
14
15   describe "GET 'next'" do
16     it "gets the next question" do
17       PEATEngine.should_receive(:next_question).and_return(@bpa
       )
18       get :next, { building_id: "building_id"}
19     end
20   end
21 end
```

El código anterior se tiene un ejemplo de una prueba unitaria para el controlador `QuestionsController` (línea 3) que maneja las acciones en relación a las preguntas asociadas al perfil de un edificio. En las líneas 4-13 se inicializan los modelos necesarios para llevar a cabo las pruebas de este controlador. En las líneas 16-20 se tiene por medio de la declaración `it` la prueba unitaria sobre la acción `next` la cual debe regresar la siguiente pregunta a contestar del perfil del edificio. En la línea 17 se define la expectativa de que el controlador llama al método `next _question` del modelo `PEATEngine` para obtener la siguiente pregunta.

Por medio de `RSpec` se tienen pruebas de los controladores y vistas principales del sistema `PEAT`. Los modelos no son probados directamente dado que los modelos hacen uso de la clase `Bezel::Base` de la biblioteca `Bezel` la cual también tiene pruebas de unidad por medio de `RSpec`.

Pruebas de aceptación

Para el sistema `PEAT` se hace uso del marco de trabajo *Cucumber* el cual permite especificar y ejecutar pruebas de aceptación siguiendo el proceso del desarrollo guiado por comportamiento.

```
1  Característica:
2  Permitir a los usuarios el añadir una recomendación a su plan
     de ahorro
3  El usuario tiene una cuenta, una dirección de servicio y un
     edificio
4
5  Antecedentes:
6  Dado que ingreso como el usuario @juan por vez primera
7  Y que tengo una cuenta, una dirección de servicio y un
     edificio
8
9  @javascript
10 Escenario: Despliega la lista de recomendaciones
11 Cuando visitó la página de recomendaciones
12 Entonces debo ver un botón de añadir al plan en cada
     recomendación
13
14 @javascript
15 Escenario: Cambiar al estado "Completado" a una recomendación
16 Dado que visitó la página de recomendaciones
17 Y selecciono el primer botón añadir al plan
18 Debo ver un menú de selección
19 Cuando selecciono la opción "Completado" en el menú
     desplegable
20 Entonces el botón de añadir al plan debe indicar "Completado"
21
22 @javascript
23 Escenario: Añadir una recomendación al plan de ahorro
24 Dado que visitó la página de recomendaciones
```

```
25  Cuando selecciono "Añadir al plan" en la primera recomendació
      n
26  Y selecciono "Guardar" en la ventana.
27  Entonces el botón de añadir al plan debe indicar "Agregado"
28  Y no debo ver el menú despegable
```

El código anterior es una prueba de aceptación sobre el requerimiento de que los usuarios puedan añadir una recomendación a su plan de ahorro. En las líneas 1-3 se documenta el objetivo de la característica que se esta probando, se prueba una sola característica por archivo pero se pueden tener varios escenarios por característica. En las líneas 5-7 se definen los antecedentes, es decir, el contexto inicial que comparten todos los escenarios. Cada escenario empieza con la declaración `Escenario:` con una breve descripción del escenario, las líneas posteriores indican tanto el contexto como la expectativa del escenario. Los escenarios pueden ser etiquetados en este ejemplo tienen la etiqueta `@javascript` lo que indica que estos escenarios hacen uso de un navegador con soporte a Javascript para ejecutar la prueba.

Cabe señalar el parecido entre el caso de uso *Administrar recomendaciones* y la prueba de aceptación, esto es una de las ventajas del uso de *Gherkin* que es el lenguaje DLE definido por *Cucumber* para facilitar la implementación de pruebas de aceptación a partir de casos de uso.

Cucumber da acceso al comando `cucumber` que permite ejecutar y verificar que el sistema pasa todas las pruebas de aceptación del sistema. El servidor *Jenkins* ejecuta el comando `cucumber` si el resultado de las pruebas unitarias es exitoso. Si el resultado de las pruebas de aceptación es positivo entonces se hace un despliegue del sistema a los ambientes de prueba.

4.2.4. Ambientes de prueba

Después de que tanto las pruebas de unidad como las de aceptación dan resultados positivos el servidor *Jenkins* realiza el despliegue del sistema al ambiente *peatstage*.

En el ambiente *peatstage* se realizan pruebas de rendimiento por medio de la herramienta CloudTest, la cual permite realizar pruebas de carga y rendimiento para sistemas web, además esta herramienta tiene integración con *Jenkins* por lo que se agregó como una etapa más en el proceso de despliegue.

Usando CloudTest se tenían tres pruebas de carga, teniendo en cada prueba 500, 1000 y 2000 usuarios concurrentes respectivamente. Si estas pruebas eran satisfactorias entonces el servidor *Jenkins* hace un despliegue del sistema en los ambientes *peattest* y *peatqa*.

4.2.5. Ciclo de desarrollo

En `PEAT` se tiene que el modelo de desarrollo es una combinación tanto del modelo en cascada y del modelo incremental, teniendo en un inicio la creación de documentos sobre los requerimientos de los componentes del sistema y su arquitectura general. Luego se aplicó el modelo incremental al realizar la implementación de los componentes del sistema `PEAT` en forma incremental por medio de iteraciones con duración

de dos semanas para cada iteración, de esta forma en cuanto se tenía suficiente información sobre los requerimientos de un componente se programaba el inicio de su implementación para la siguiente iteración.

Durante el desarrollo se hizo uso de prácticas ágiles como:

- Reunión diaria de sincronización del equipo.
- Desarrollo guiado por pruebas.
- Planeación por iteraciones.
- Póker de planeación.
- Retrospectivas al final de una iteración.
- Despliegue continuo.

En el sistema `PEAT` se tenía el siguiente ciclo de desarrollo de software:

1. El programador crea una nueva rama de trabajo en el repositorio `Git` para la implementación de una nueva característica del sistema.
2. Conforme se van acumulando los cambios del código en la rama de trabajo el servidor *Jenkins* ejecuta solamente las pruebas de unidad y aceptación, permitiendo una rápida retroalimentación del estado del sistema al programador.
3. Cuando la implementación se considera terminada y las pruebas son positivas, se realiza una revisión del código de la rama de trabajo por el resto del equipo.
4. Después de la revisión y dado el visto bueno del equipo se integra el código de la rama de trabajo a la rama «master», lo cual provoca el inicio de todo el proceso de despliegue.
5. El servidor ejecuta las pruebas de unidad y aceptación del sistema en caso positivo realiza un despliegue al ambiente *peatstage*, en caso contrario el equipo es informado de cuales pruebas fallaron.
6. El servidor ejecuta las pruebas de carga usando el ambiente *peatstage* si el resultado es positivo se hace un despliegue a los ambientes *peattest* y *peatqa*, en caso contrario se informa al equipo sobre que partes del sistema no tiene el rendimiento esperado.
7. En coordinación con PG&E se despliega la versión en uso en *peattest* al ambiente en producción.

Cabe mencionar que al menos cada semana un representante de PG&E se presentaba en las instalaciones de C3 Energy para discutir y hablar sobre los avances del sistema, aparte se tenia una comunicación electrónica continua con los resultados de las pruebas realizadas por PG&E en los ambientes de pruebas.

Cualquier falla se convertía en una tarea prioritaria para el equipo de desarrollo para evitar que el lapso de tiempo en que el sistema no puede ser desplegado sea el mínimo posible.

Haciendo uso del despliegue continuo el equipo entero tiene conocimiento sobre el estado del sistema aunque el sistema PEAT se encuentre en un estado inconsistente por fallas, ya sea en las pruebas unitarias, aceptación o de carga, se tiene información sobre la naturaleza y origen de las fallas. Esta información permite al equipo arreglar el sistema de una forma eficiente.

Conclusiones

En este documento se expuso el diseño e implementación del sistema PEAT que permite a sus usuarios el identificar y monitorear su consumo energético. En C3 Energy se tienen varios sistemas en producción haciendo uso de lenguajes de programación como Java y Javascript y herramientas como Rhino y Ext JS.

En la implementación de PEAT se presentaron nuevas retos como la necesidad de una interfaz simple y de rápida respuesta a las acciones del usuario y dar servicio a un número considerable de usuarios. Estos requerimientos no se podían solucionar de forma óptima haciendo uso de los lenguajes y herramientas usadas hasta ese momento. Es por eso que al iniciar la implementación del sistema PEAT se eligió tener un ambiente de desarrollo multi-lenguaje en el cual se hace uso de un lenguaje de programación o herramienta en los contextos en que éstos dan la mayor ventaja. Un ambiente multi-lenguaje trae consigo sus propios retos como la integración con sistemas ya existentes y el despliegue exitoso de los sistemas a producción.

Haciendo uso de las capacidades de metaprogramación del lenguaje de programación Ruby y de la creación de un lenguaje de dominio específico en la biblioteca Bezel se facilitó la implementación de los modelos necesarios para el funcionamiento del sistema de forma eficiente y con la flexibilidad necesaria para lograr la integración del sistema PEAT con los subsistemas ya existentes, permitiendo al programador definir modelos y sus asociaciones con pocas líneas de código.

El uso de lenguajes de dominio específico fue el aspecto más sobresaliente para el desarrollo exitoso del sistema PEAT por su ayuda en la integración con los subsistemas ya existentes y en la automatización de la configuración y despliegue del sistema.

El uso del despliegue continuo y de pruebas con usuarios finales fueron vitales para que el sistema cumpliera con su objetivo de dar la mayor utilidad posible a los usuarios PyMES sobre su consumo energético. La retroalimentación continúa obtenida por los ambientes de prueba permitió detectar deficiencias tanto en el backend como en la interfaz del usuario, siendo la principal deficiencia detectada la dificultad de ingreso de la información inicial para crear un perfil de un edificio. Gracias a esta retroalimentación se pudo obtener una interfaz mas intuitiva para el usuario sin afectar el tiempo de entrega.

También por medio del despliegue continuo se logró tener un lanzamiento a la etapa de producción sin problemas, dado el que con anterioridad se realizaron varios despliegues en ambientes de prueba los cuales eran idénticos al ambiente de producción.

El sistema PEAT fue recibido muy bien por los usuarios y el cliente PG&E ya que desde hace cuatro años se encuentra en funcionamiento dando servicio a cerca de 250,000 usuarios PyMES a la fecha[20]. Su éxito provocó que el sistema fuera también adoptado por otras compañías proveedoras de electricidad y gas como San Diego Gas & Electric[7] y Southern California Edison, que prácticamente abarca el estado de California en los Estados Unidos de América[4].

Bibliografía

[1] S. Barrett y Byrnes. The ssh protocol. URL: http://www.snailbook.com/protocols.html (visitado 08-04-2016).

[2] K. Beck. *Test-driven development*. Addison-Wesley, 1.ª ed., 2003, págs. 9-10.

[3] D. Chelimsky. *The rspec book*. Pragmatic, 2010, págs. 3-6.

[4] C. E. Commission. California energy maps. URL: http://www.energy.ca.gov/maps/serviceareas/electric_service_areas.html (visitado 01-05-2016).

[5] Datastax. Advanced time series with cassandra. URL: http://www.datastax.com/dev/blog/advanced-time-series-with-cassandra (visitado 01-05-2016).

[6] R. Documentation. Crud, verbs, and actions. URL: http://guides.rubyonrails.org/routing.html#crud-verbs-and-actions (visitado 19-10-2015).

[7] C. Energy. C3 energy announces fiscal year 2014 results. URL: http://www.reuters.com/article/ca-c3-energy-idUSnBw146592a+100+BSW20140514 (visitado 01-05-2016).

[8] D. Flanagan e Y. Matsumoto. *The ruby programming language*. O'Reilly, 2008, págs. 2, 266-278.

[9] T. A. S. Foundaiton. The apache cassandra project. URL: http://cassandra.apache.org (visitado 23-10-2015).

[10] M. Fowler. Continuous delivery. URL: http://martinfowler.com/bliki/ContinuousDelivery.html (visitado 23-03-2016).

[11] M. Fowler. Gui architectures. URL: http://martinfowler.com/eaaDev/uiArchs.html#ModelViewController (visitado 23-10-2015).

[12] E. Gamma. *Design patterns*. Addison-Wesley, 1.ª ed., 1995, págs. 4-6.

[13] J. Humble y D. Farley. *Continuous delivery*. Addison-Wesley, 2011.

[14] S. Krishnamurthi. *Programming languages: application and interpretation*. Shriram Krishnamurthi, 2003, págs. 316-317.

[15] R. Leonard. *Restful web services*. O'Reilly, 1.ª ed., 2007, págs. 79-105.

[16] M. S. Mikowski y J. C. Powell. *Single page web applications*. Manning, 1.ª ed., 2014, págs. 3-5.

[17] M. D. Network. Rhino. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino (visitado 23-10-2015).

[18] P. Perrotta. *Metaprogramming ruby*. Pragmatic Bookshelf, 1.ª ed., 2010, págs. 6-9.

[19] PG&E. Rates - time of use. URL: http://www.pge.com/en/mybusiness/rates/tvp/toupricing.page (visitado 19-10-2015).

[20] PG&E. *Smart grid annual report - 2013*. PG&E, 2013, pág. 25.

[21] N. Rappin. *Rails test prescriptions*. Pragmatic Bookshelf, 2010, págs. 1-10.

[22] S. Ruby, D. Thomas y D. H. Hansson. *Agile web development with rails 4*. Pragmatic Bookshelf, 1.ª ed., 2013, págs. 29-34.

[23] I. Sommerville. *Software engineering*. Pearson, 9.ª ed., 2011, pág. 734.

[24] W3C. Web services glossary. 2004. URL: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice (visitado 19-10-2015).