Oscar Oranagwa Seguir
:)
9 de enero · 18 minutos de lectura

Cómo construí un servidor de CI usando Docker

Introducción

La integración continua , CI, se está convirtiendo rápidamente en una parte arraigada de cada equipo de ingeniería. Y de las diversas prácticas de CI, las compilaciones de autoevaluación a menudo sirven como foco central para muchos proyectos, por lo tanto, servicios populares como Circle CI, HerokuCI , T ravis CI, S emaphore , AWS Codebuild etc. han crecido para proporcionar integraciones fáciles para afirmar el confiabilidad de un cambio de código dado según lo determinado por las pruebas adjuntas. Estas integraciones a menudo involucran un servidor de CI que monitorea constantemente el repositorio de código de proyecto y los cambios en el repositorio, establece un entorno imparcial para afirmar el estado del proyecto con el cambio que ocurrió.

Aquí hay un extracto de Thoughtworks sobre cómo se ve un flujo típico de integración continua:

- Los desarrolladores revisan el código en sus espacios de trabajo privados
- Cuando termine, envíe los cambios al repositorio
- El servidor de CI supervisa el repositorio y verifica los cambios cuando ocurren
- El servidor de CI construye el sistema y ejecuta pruebas de unidades y de integración
- El servidor de CI libera artefactos desplegables para probar
- El servidor de CI asigna una etiqueta de compilación a la versión del código que acaba de crear
- El servidor de CI informa al equipo de la creación exitosa o fallida

Muchas veces, surgen situaciones en las que existe un desacuerdo entre los resultados de la prueba en la máquina del desarrollador y el entorno del servidor de CI. Las investigaciones sobre ese desacuerdo a menudo desentrañan las disparidades en ambos entornos que dan lugar a los elementos causales. Mientras realizo muchas de estas investigaciones, me encuentro cada vez más interesado en lo que sucede entre bastidores en estos servidores de CI y en los diversos (posibles) elementos que se unen para formar un servidor de CI. Y con una curiosidad creciente y un sentido de la aventura, me embarqué en un viaje para (re) crear uno de esos servidores.

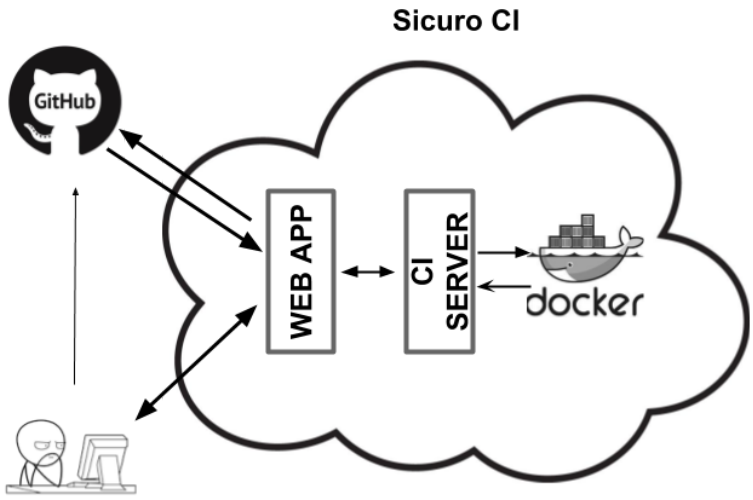
Este artículo identifica las diversas partes de un servidor confiable de CI y explora cómo juntarlas usando docker.

Visión de conjunto

Un servicio de CI típico tendrá una interfaz para que los usuarios inicien sesión y conecten las cuentas de control de versiones de su proyecto, como GitHub y Bitbucket. Una vez conectado, el usuario debería ser capaz de especificar proyectos dentro de estas cuentas que el servidor de CI monitorearía para eventos de cambio de código tales como confirmaciones y solicitudes de extracción. Una vez que ocurra cualquiera de estos cambios, el CI probará el cambio para determinar su integrabilidad con la base del código.

Implementación

La implementación se centra principalmente en GitHub, pero el resultado final se puede ampliar fácilmente para trabajar con cualquier otro servicio de alojamiento de repositorio de control de versiones Git basado en la web.



La aplicación web proporciona una interfaz donde los usuarios inician sesión usando GitHub OAuth, permitiendo que el servidor de CI tenga acceso para ver sus proyectos de GitHub. Los usuarios reciben una lista de sus proyectos y pueden seleccionar los proyectos que el servidor de CI debe monitorear. El servidor de CI registra un webhook en GitHub para eventos comunes (solicitudes de extracción y confirmación) en los proyectos seleccionados para que cada vez que ocurra alguna de estas acciones, GitHub notifique al servidor de CI. Y al recibir tales notificaciones, el servidor de CI extrae el último cambio de código en el proyecto y ejecuta las pruebas adjuntas con la salida y el progreso visualizados en una interfaz para que el usuario pueda verlos.

Este tutorial de implementación hace alusión fácilmente a dos secciones principales para un servidor de CI:

- Una sección para la interacción directa del usuario: inicio de sesión, suscripción de proyecto, visualización del progreso de la prueba. es decir, estas partes requieren la presencia del usuario para entrar en acción
- Una sección para extraer el código del proyecto, ejecutar las pruebas y registrar los resultados; es decir, no requiere la entrada del usuario (si el usuario está en línea o no) para hacerlo.

La primera parte, me refiero a la aplicación web y la segunda como el CI y juntos forman el servidor de CI (que yo llamo **Sicuro**).

El CI

El único propósito del servidor de CI será ejecutar las pruebas para un proyecto determinado y proporcionar información sobre el progreso y el resultado de la prueba. Para realizar esta tarea, se requerirá información tal como:

- El nombre del proyecto
- La URL del proyecto (en GitHub)
- La rama del proyecto de destino o hash de confirmación
- El lenguaje del proyecto (es decir, Ruby, Javascript, Golang, Python ...)

Una alternativa al lenguaje del proyecto sería el comando exacto para ejecutar la prueba, pero tal como se explica en la sección *Contenedores de prueba* (a continuación), encontraríamos que el lenguaje es más útil.

Con esta información, el servidor de CI configura el entorno de acoplador necesario para ejecutar las pruebas para el proyecto determinado. Las siguientes secciones tienen una descripción detallada del entorno de la ventana acoplable.

Ahora, con el contenedor docker fielmente avanzando, ejecutando las pruebas, todo lo que queda es proporcionar información sobre el progreso del proceso de prueba. Una solución rápida es canalizar la salida de los registros del contenedor de la prueba a un archivo y luego rastrear el archivo para obtener nueva información de registro.

Poniendo todo esto junto con Golang, nuestro servidor ci debería parecerse a:

```
paquete ci

importación (
        " fmt "
        " log "
        " os "
        " os / exec "
        " ruta / ruta de archivo "
        " cuerdas "

        " github.com/kjk/betterguid "
)

const (
        // LogFileExt es la extensión de archivo utilizada
        LogFileExt = " .log "
)

var (
        // ciDIR es la ruta absoluta al directorio de CI
        ciDIR = ruta de archivo Únete (os. Getenv ( " ROOT
        // LogDIR es la ruta absoluta al directorio de reg
        LogDIR = ruta de archivo. Únete (ciDIR, " registro
)

// JobDetails contiene la información necesaria para ejecu
tipo  JobDetails  struct {
        // LogFileName es el nombre que se utilizará para
        // Siempre debería ser el hash de confirmación p.
        // o nombre de la rama, por ejemplo, maestro
         Cadena LogFileName
cadena  logFilePath
        // ProjectRespositoryName es el nombre del reposit
        // Se usa cuando se clona el proyecto en un conten
        ProjectRespositoryName  string
        // ProjectBranch es la rama de destino para ejecut
        // También podría ser un hash de confirmación si e
         Cadena ProjectBranch
        // ProjectRespositoryURL es la url SSH para extrae
         Cadena ProjectRepositoryURL
        // ProjectLanguage es el lenguaje de programación
        // Esto se usaría para determinar la imagen del ac
         Cadena de ProjectLanguage
        // UpdateBuildStatus es una función de devolución
```

```
45          // se ejecutaría con el estado de construcción pen
46          // Una vez que se inician las pruebas, se ejecuta
47          // Al finalizar la prueba, se volvería a ejecutar
48          UpdateBuildStatus  func ( cadena )
49  }
50
51  func  init () {
52          // establecer la variable de entorno de ruta ci
53          os. Setenv ( " CI_DIR " , ciDIR)
54  }
55
56  // Run activa el servidor de CI para el trabajo dado
57  // Construye la ruta absoluta al archivo de registro de tr
58  // Termina si una rutina está actualmente activa para el t
59  // De lo contrario, configura una nueva rutina para el tra
60  func  Run ( job * JobDetails ) {
61          trabajo. logFilePath = ruta de archivo. Join (LogD
62          err  : =  createDirFor ( job.logFilePath )
63          si err! = nil {
64                  pánico (err)
65          }
66
67          // asegurar que el archivo no se está escribiendo
68          si  ActiveCISession ( job.logFilePath ) {
69                  Iniciar sesión. Println ( " Un trabajo est
70                  regreso
71          }
72
73          // preparar el archivo de registro, es decir, borr
74          si  err  : = exec. Comando ( " bash " , " -c " , "
75                  Iniciar sesión. Printf ( " Error: % s se p
76                  regreso
77          }
78
79          trabajo. ProjectLanguage = cadenas. ToLower (job.
80          ir a  correrCI (trabajo)
81  }
82
83  func  createDirFor ( nomArchivo  cadena ) de error {
84          dir , file  : = filepath. Split (fileName)
85          Iniciar sesión. Printf ( " Hacer dir: % s para el
86          devolver os. MkdirAll (dir, 0755 )
87  }
88
89  func  runCI ( job * JobDetails ) {
```

```
90      Iniciar sesión. Printf ( " Trabajo en ejecución: % v \ n
```

Notas:

1. La `JobDetails` estructura detalla la información requerida para ejecutar el servidor. Una adición interesante es la `updateBuildStatus` función de devolución de llamada, que se activa al final de la prueba con el estado de la prueba. Encontraremos su uso más adelante en la aplicación web.
2. El `Run` func sirve como el principal punto de entrada y sus responsabilidades incluyen:

- Creando el archivo de registro (si no existe)
- Asegurar que no haya trabajos activos de CI escribiendo en dicho archivo de registro (`lsof filename`)
- Configuración de un trabajo de CI para ejecutar las pruebas. Los trabajos se ejecutan en rutinas para permitir el manejo concurrente de solicitudes.

3. Vincula el contenedor a posibles servicios a pedido, como la base de datos, y configura las variables de entorno necesarias para conectarse a ellos. Para simplificar, estos servicios se ejecutarán constantemente y aprovecharán las redes de contenedor docker , conectaremos los contenedores de prueba a los recursos según sea necesario añadiendo los contenedores a la `ci_default` red.

El comando final ejecutado para ejecutar la prueba sería algo de la forma:

```
docker run -it --rm \
   -v [RUTA PATH-A-SSH-FOLDER]: /. ssh \
   -e PROJECT_REPOSITORY_URL = [THE-PROJECT-REPO-URL] \
   -e PROJECT_REPOSITORY_NAME = [NOMBRE-PROYECTO] \
   -e PROJECT_BRANCH = [PROJECT-TARGET-BRANCH] \
   -e DATABASE_URL = [POSTGRES-DATABASE-URL] \
   -e REDIS_URL = [REDIS-DATABASE-URL] \ -
   network ci_default \
   [DOCKER_IMAGE_REPO] / sicuro_ [PROJECT-LANGUAGE ]
```

Dado un proyecto de rieles de muestra con detalles del trabajo

```
{
  LogFileName: "0sc / activestorage-cloudinary-service /
master",
  ProjectRepositoryURL: "git@github.0sc/activestorage-
cloudinary-service",
  ProjectBranch: "master",
  ProjectLanguage: "ruby",
  ProjectRespositoryName: "activestorage-cloudinary-
servicio ",
}
```

El comando sería

```
docker run -it --rm \
   -v [RUTA PATH-A-SSH-FOLDER]: /. ssh \
   -e PROJECT_REPOSITORY_URL=git@github.0sc/activestorage-
cloudinary-service \
   -e PROJECT_REPOSITORY_NAME = activestorage-cloudinary-
servicio \
   -e PROJECT_BRANCH = master \
   -e DATABASE_URL = postgres: // postgres @ postgres: 5432
/ postgres \
   -e REDIS_URL = redis: // redis: 6379 \
   --network ci_default \
   xovox / sicuro_ruby: 0.2
```

Contenedores de prueba

El entorno requerido para ejecutar la prueba difiere para diferentes proyectos. Una diferencia obvia es el tiempo de ejecución requerido según el lenguaje de programación del proyecto. Un proyecto de Ruby tiene una dependencia de tiempo de ejecución diferente de un proyecto de Javascript que es diferente para un proyecto de Python, etc. Y estas diferencias deben ser tenidas en cuenta por el servidor de CI.

Una forma común de manejar esto es tener una imagen de "dios" preconstruida que esté configurada con todas las dependencias básicas para cada idioma. Del mismo modo que cualquier máquina de desarrollo puede configurarse para ejecutar proyectos en diferentes idiomas, también los contenedores de prueba se pueden preparar con todas las necesidades para ejecutar proyectos en todos los idiomas. Básicamente una imagen de acoplador con el tiempo de ejecución necesario y las dependencias para cada lenguaje (compatible): Ruby, Javascript, PHP, etc. Este enfoque tiene la ventaja obvia de que elimina la carga de las preocupaciones y ajustes específicos del idioma, y es la implementación preferida para muchos de los CI's populares disponibles.

Para esta implementación, sin embargo, decidí tener imágenes separadas para cada idioma. Esta decisión es ayudar a aprender las complejidades del entorno de cada idioma y descifrar los enredos de dependencia que vienen con varios proyectos.

Los archivos docker para los diferentes idiomas se pueden encontrar aquí . Aquí está el archivo docker de ruby:

```
DESDE ubuntu: 16.04

# paquetes necesarios para la construcción de rubíes con rv
Ejecutar apt-get update && -qqy apt-get install -qqy \
        bzip2 \
        gawk \
        g ++ \
        gcc \
        hacer \
        libreadline6-dev \
        libyaml-dev \
        libsqlite3-dev \
        sqlite3 \
        autoconf \
        libgmp-dev \
        libgdbm-dev \
        libncurses5-dev \
        automake \
        libtool \
        bisonte \
        pkg-config \
        libffi-dev \
        git \
        curl \
        nodejs \
        tzdata \
        libpq-dev \
        libmysqlclient-dev \
        qt5-default \
        libqt5webkit5-dev \
        imagemagick \
        libmagickwand-dev \
                jq \
        ssh \
        xvfb \
        && rm -rf / var / lib / apt / lists \
        && truncado -s 0 / var / log / * log

```

Y como puede haber observado en el código de CI, se espera que cada imagen resultante esté etiquetada, `sicuro_[language]` por ejemplo `sicuro_ruby`

Pero independientemente del idioma de la imagen, todos comparten una `docker-entrypoint.sh` secuencia de comandos similar .

El punto de entrada de Docker

```
#! / bin / bash
trampa  ' salida ' ERR

echo  " <h3> Iniciando la compilación </ h3> "

echo  " <h3> Agregar claves SSH </ h3> "
mkdir -p /root/.ssh/ && cp -R .ssh / *  " $ _ "
chmod 400 /root/.ssh/ *  && \
    ssh-keyscan github.com > /root/.ssh/known_hosts && \
    ssh-keyscan bitbucket.com >> /root/.ssh/known_hosts

echo  " <h3> Código fuente de pago </ h3> "
git clone $ {PROJECT_REPOSITORY_URL}  $ {PROJECT_REPOSITORY
cd  $ {PROJECT_REPOSITORY_NAME}
git checkout $ {PROJECT_BRANCH}
# Haz que rvm vuelva a verificar la versión de ruby
cd  .

echo  " <h3> Dependencias </ h3> "

# dependencias de idioma predeterminadas
echo Exportando RAILS_ENV
exportar RAILS_ENV = prueba
eco Exportando RACK_ENV
exportar RACK_ENV = prueba
echo Exportando SECRET_KEY_BASE
```

Este script orquesta el flujo de ejecución de prueba. Se inicia agregando algunas claves ssh genéricas para clonar proyectos de GitHub. Y dado que solo estamos interesados en repositorios de acceso público, cualquier combinación de clave ssh debidamente registrada en GitHub debería funcionar. (Sin embargo, para permitir el acceso a repositorios privados, esto se extendería fácilmente para permitir a los usuarios proporcionar claves ssh específicas para sus cuentas de GitHub).

Con las teclas SSH establecidas, procede a clonar el proyecto. Recuerde que el `PROJECT_REPOSITORY_URL` y el `PROJECT_REPOSITORY_NAME` son parte de la información proporcionada al iniciar el contenedor. Esta

información se usa aquí para clonar el proyecto en una carpeta con el nombre del proyecto. Para nuestro trabajo de muestra, esto da como resultado:

```
git clone $ {PROJECT_REPOSITORY_URL} $
{PROJECT_REPOSITORY_NAME}
# git clone git@github.0sc/activestorage-cloudinary-service
activestorage-cloudinary-service
```

Ahora tenemos el proyecto, `cd` en la carpeta y `checkout` en la rama deseada. Recuerde `PROJECT_BRANCH` , ¿verdad?

```
cd activestorage-cloudinary-service
git checkout master
```

En este punto, podríamos establecer algunas variables de entorno específicas del lenguaje estándar. Por ejemplo, un proyecto de rieles se beneficiaría de saber `RAILS_ENV` y / o `RACK_ENV` en qué proyecto se está ejecutando y también los `SECRET_KEY_BASE` valores para el entorno dado. Lo mismo para una aplicación NodeJS con `NODE_ENV` .

Este también sería un buen momento para configurar algunos ajustes específicos de la máquina, como iniciar un navegador sin cabeza para pruebas de integración de carpincho, etc. El objetivo es que al final de esto, todas las configuraciones específicas de la máquina estén en su lugar.

Ahora volvemos nuestra atención a la configuración predeterminada del proyecto para el idioma. La creación de una aplicación comienza con NodeJS `npm install` , una aplicación Python: `pip install` , `bundle install` para una aplicación de Ruby ... Podría haber otra configuración estándar necesaria requerida similares; configurando el DB y ejecutando migraciones ... etc. Con todo esto cuidado, finalmente estamos listos para ejecutar el comando de prueba estándar para el lenguaje del proyecto.

Aquí hay una descripción general de algunos valores predeterminados para Ruby on Rails y NodeJS

Idioma	Dependencias de la máquina	Configuración del proyecto	Prue
Ruby / Rails	RAILS_ENV RACK_ENV	bundle install bundle exec	prue de r ejec

Hasta ahora, siempre que un proyecto determinado se adhiera a sus valores predeterminados de idioma, nuestro servidor de CI podría manejarlo con éxito. Pero desafortunadamente, muy pocos proyectos lo hacen.

La historia de sicuro.yaml

Al igual que todos los CI (buenos), dejando de lado los valores predeterminados de secuencias de comandos, debe haber opciones para personalizar cada uno de los procesos de ejecución de pruebas (es decir, las dependencias de la máquina, la configuración, las fases de prueba) por proyecto. Los proyectos a menudo encuentran la necesidad de más de las ofertas predeterminadas del lenguaje del proyecto; un marco de prueba alternativo (en `rspec` lugar de `minitest` , ruby), un administrador de dependencias alternativo (en `glide` lugar de `d` , Golang), un corredor de tareas (`gulp` , NodeJS) etc. Acomodar estas variaciones da lugar a `circle.yml` , `travis.yml` ... utilizado por las respectivas principales ofertas de CI. Y en nuestro caso, `sicuro.json`

`Sicuro.json` es un archivo de configuración opcional que se puede incluir en la base de cualquier proyecto para la personalización / anulando las diversas etapas: *dependencias* , *configuración* y *prueba* secciones del proceso de ejecución de la prueba de CI. Si el archivo está presente, los contenidos se leen y se tienen en cuenta al ejecutar las pruebas:

Aquí hay una muestra

```
{
    " dependencias " : {
        " anular " : falso ,
        " personalizado " : [
            " Exportar DB_URL = $ DATABASE_URL "
        ]
    },
    " configuración " : {
        " anular " : falso ,
        " personalizado " : [
            " apt-get install -y phantomjs " ,
            " instalación de bower "
        ]
    },
    " prueba " : {
```

Para simplificar, la personalización se limita a dos posibilidades:

- Las ejecuciones predeterminadas se pueden desactivar para el proyecto configurando: `override: true`
- Y las ejecuciones personalizadas también se pueden incluir para un proyecto

Ambas personalizaciones no son mutuamente excluyentes. Por lo tanto, supongamos que un proyecto nodejs determinado `phantomJS` debe incluirse en la configuración y lo usa `grunt` ; el ejemplo anterior sería un `sicuro.json` archivo válido para el proyecto.

El `sicuro.json` archivo se analiza usando la ingeniosa `jq` biblioteca; "Un procesador JSON liviano y flexible de línea de comandos". JQ es una herramienta increíble y entre sus muchos beneficios, para nuestro caso de uso, su operador alternativo , `//,` es útil. Permite definir retrocesos para claves faltantes en una secuencia de comandos json que aprovechamos para hacer que las distintas claves en el `sicuro.json` archivo sean opcionales.

Al lanzar el `sicuro.json` análisis `jq` en la mezcla, la lógica de ejecución para cada una de las secciones de prueba se convierte en

- Si `sicuro.json` está presente
- Si se establece el indicador de anulación, omita el comando predeterminado

- Si el indicador de anulación no está configurado. Realice los comandos predeterminados
- Si se establece el indicador personalizado, realice los comandos personalizados

Con esto, la `docker-entrypoint.sh` secuencia de comandos se actualiza a

```
#! / bin / bash
trampa  ' salida ' ERR

echo  " <h3> Iniciando la compilación </ h3> "

echo  " <h3> Agregar claves SSH </ h3> "
mkdir -p /root/.ssh/ && cp -R .ssh / *  " $ _ "
chmod 400 /root/.ssh/ *  && \
    ssh-keyscan github.com > /root/.ssh/known_hosts && \
    ssh-keyscan bitbucket.com >> /root/.ssh/known_hosts

echo  " <h3> Código fuente de pago </ h3> "
git clone $ {PROJECT_REPOSITORY_URL}  $ {PROJECT_REPOSITORY
cd  $ {PROJECT_REPOSITORY_NAME}
git checkout $ {PROJECT_BRANCH}
# Haz que rvm vuelva a verificar la versión de ruby
cd  .

# verificar si sicuro.json está presente
SICURO_CONFIG_PRESENT = falso
SICURO_CONFIG_FILE =. / Sicuro.json

si [ -r ./sicuro.json] ;  entonces
    SICURO_CONFIG_PRESENT = true
fi

echo  " <h3> Dependencias </ h3> "

si  ! ( $ SICURO_CONFIG_PRESENT  &&  $ ( cat $ SICURO_CONFI
    # dependencias de idioma predeterminadas
    echo Exportando RAILS_ENV
    exportar RAILS_ENV = prueba
    eco Exportando RACK_ENV
    exportar RACK_ENV = prueba
    echo Exportando SECRET_KEY_BASE
    exportar SECRET_KEY_BASE = some-random-string-will-suff
    # iniciar Xvfb controlador sin cabeza
    Xvfb: 99 y  exportación DISPLAY =: 99
fi
si  $ SICURO_CONFIG_PRESENT  ;  entonces
    source  <( cat $ SICURO_CONFIG_FILE  | jq --raw-output
```

En una palabra:

- El servidor ci recibirá una solicitud de trabajo que contiene el nombre del proyecto, url, idioma
- Continúa solo si no se está ejecutando ningún otro trabajo de la misma firma.
- Establece las variables de entorno necesarias para que los recursos, como la base de datos, estén disponibles para el entorno de prueba
- Gira un contenedor acoplable basado en el idioma del proyecto con los detalles necesarios para ejecutar la prueba y adjunta las salidas del registro del contenedor al archivo de registro especificado
- Los scripts del punto de entrada del contenedor configuran las claves ssh necesarias y proceden a clonar el proyecto
- Comprueba si el proyecto tiene algún `sicuro.json` archivo
- Configure dependencias de máquina específicas para el proyecto teniendo en cuenta los contenidos y las instrucciones en el `sicuro.json` archivo (si está presente)
- Configure las dependencias específicas del proyecto para el proyecto teniendo en cuenta los contenidos y las instrucciones en el `sicuro.json` archivo (si está presente)
- Ejecuta la prueba para el proyecto teniendo en cuenta el contenido y las instrucciones en el `sicuro.json` archivo (si está presente)

La aplicación web

Hasta ahora hemos explorado las partes internas del servidor de CI y, como es de esperar, los usuarios no deberían interactuar con las especificaciones básicas del servidor. La aplicación web existe para abstraer estas interacciones al proporcionar una experiencia más rica para usar el servidor.

La aplicación web atiende a dos clientes principales, usuarios (humanos), que vienen a suscribir sus proyectos para que el servidor CI los supervise y los servidores *VCS* (sistema de control de versiones) (como GitHub) donde se alojan estos proyectos.

La interfaz de usuario

Para que el servidor de CI supervise un proyecto, el usuario deberá suscribirse al proyecto. La suscripción de un proyecto requiere que el

usuario permita que el servidor de CI registre un webhook con el servidor de *vc en* línea del proyecto . La mayoría de los *VCS* proporcionan webhooks, que permiten que las aplicaciones se suscriban a ciertos eventos en el proyecto. Y cuando uno de esos eventos se desencadena, el vcs envía una carga a la URL configurada del webhook.

El primero en configurar esto es tener una integración de OAuth que permita a los usuarios iniciar sesión con su cuenta de GitHub. El flujo de inicio de sesión está lleno de la autorización necesaria para que la aplicación web lea y enumere los proyectos de los usuarios (públicos) en GitHub.

Este tutorial detalla cómo registrar una aplicación GitHub OAuth y obtener sus códigos `GITHUB_CLIENT_ID` *y* `GITHUB_CLIENT_SECRET` *. El proceso de registro requerirá una URL de devolución de llamada que puede establecer en localhost por ahora. Más adelante en la* `hosting` *siguiente sección, actualizaremos esto a la URL de AWS para nuestra aplicación alojada.*

Nota : Si desea ampliar el acceso a repositorios privados, además de incluirlos en el permiso solicitado, como se mencionó anteriormente, también deberá ajustar la sección SSH en la sección de redactores del portador para utilizar claves específicas preaprobadas por el usuario. Por lo tanto, parte de la suscripción podría ser que el usuario agregue esta clave a su cuenta de GitHub para obtener el acceso necesario y actualizar los contenedores de docker para usar dichas claves para los proyectos del usuario (consulte la sección sobre `Handling Creds` con S3).

Ahora, con acceso a los proyectos del usuario, podemos comenzar a explorar las ofertas expansivas de la API del proyecto GitHub . De interés principal en este punto es el repositorio webhook api que proporciona información sobre la suscripción a webhooks para un repositorio de proyecto determinado. Con esta información comprobamos

- Si el proyecto aún no está suscrito, brinde al usuario la opción de suscribirse
- Si el proyecto ya está suscrito, brinde al usuario la opción de ver las compilaciones de prueba para el proyecto

También podríamos incluir una opción para darse de baja, pero no hay nada de divertido por el momento :)

Aquí está el fragmento de código para lograr esto

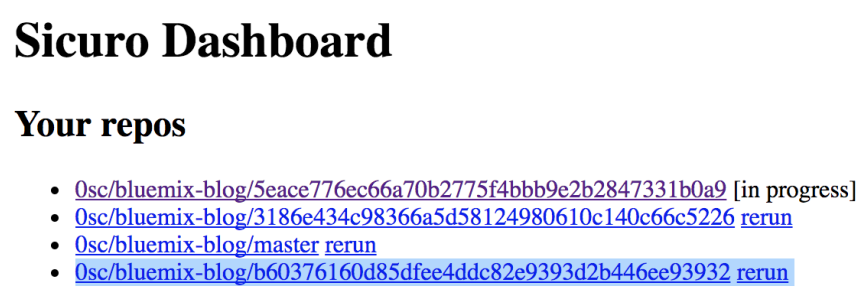
```
importación (
        " contexto "
        " log "

        " github.com/google/go-github/github "
)

// IsRepoSubscribed comprueba si el repositorio dado tiene
func  IsRepoSubscribed ( repoName  string ) bool {
        cliente  : = github. NewClient ( nil )
        ganchos , _ , err  : = cliente. Repositorios . List
                contexto. Fondo (),
                " usergithubname o id " ,
                repoName,
                & github. ListOptions {},
        )

        si err! = nil {
                Iniciar sesión. Printf ( " Error % s se pro
                devolver  falso
        }

        // para cada gancho en el cheque repo si es nuestro
        ourCallbackURL  : =  " / our / webhook / callback /
        para  _ , hook  : =  range hooks {
                si  hasActiveWebhook (hook, ourCallbackURL)
                        devolver  verdadero
                }
        }
```

A continuación, puede recorrer fácilmente todo el repositorio del usuario y evaluar `IsRepoSubscribed` para cada uno de ellos

Los resultados de compilación de prueba para un proyecto son una agregación de los registros de prueba. Un proyecto tendrá múltiples versiones correspondientes a varias confirmaciones, solicitudes de extracción y otros eventos que ocurren en el proyecto. Y una manera simple de almacenar esto adecuadamente (ya que estamos usando almacenamiento basado en archivos) es usar el hash de confirmación como el nombre del archivo de registro. Cada compilación podría contener información como el estado de la compilación, la hora, el

autor, el hash ... etc. La API de GitHub proporciona una gran variedad de detalles para elegir.

¿Recuerdas el `ActiveSession` func del servidor de CI? Aprovechamos aquí para mostrar al usuario si una construcción está actualmente en progreso. Para compilaciones completas, brindamos la oportunidad al usuario de ejecutar nuevamente la compilación y una opción para ver los resultados de la compilación de prueba.



El último bit es mostrarle al usuario la salida de registro real del contenedor y la parte genial, actualizaciones en tiempo real de una construcción en progreso. Para hacer eso, aprovecharemos una conexión WebSocket. El marco web de Gorilla para Golang viene con una muestra de websocket que es una implementación sencilla de lo que necesitamos.

Una vez que se establece la conexión de websocket, se establece un bucle infinito que supervisa el archivo de registro determinado para las modificaciones (verificando la `last modified at` hora). Y cada vez que se modifica el archivo, envía esta información al cliente.

El código del lado del cliente js colaborador es:

```
1  < pre id = " fileData " > {{. Datos }} < / pre >
2  < script type = " text / javascript " >
3    ( función () {
4      var data =  document . getElementById ( " fileData " );
5      var conn =  new  WebSocket ( " ws: // {{.Host}} / ws /
6      conn . onclose  =  function ( evt ) {
7        datos . innerHTML  =  ' Conexión cerrada ' ;
8      }
9      conn . onmessage  =  function ( evt ) {
10       datos . innerHTML  =  evt . datos ;
```

Todo bien :)

La interacción VCS

Además de autenticar a los usuarios, tenemos otras 3 empresas con los servidores VCS

- Registrar webhooks para eventos que nos interesan en los proyectos
- Recepción de la carga útil del evento webhook
- Comunicando el progreso y los resultados de la compilación de prueba

GitHub tiene una amplia documentación sobre todo lo que hay en su API webhook. Para configurar un webhook , los requisitos son enviar una `POST` solicitud al `/repos/:owner/:repo/hooks` con una carga útil, incluido el evento (s) para el que se desencadena el enlace. Esta funcionalidad junto con otras características a menudo están disponibles de manera inmediata con GitHub SDK para diferentes idiomas. La biblioteca de Golang lo tiene todo arreglado con la llamada de función:

```
...
cliente  : = github. NewClient ( nil )
gancho  : = github. Hook {
  Nombre : github. Cadena ( " web " ),
  Activo : activo,
  Eventos : [] cadena { " push " , " pull_request " },
  Config : map [ string ] interface {} {
    " content_type " : " json " ,
    " url " :           " http: // localhost / our / webhoo
    " secreto " :        " some-security-token " ,
```

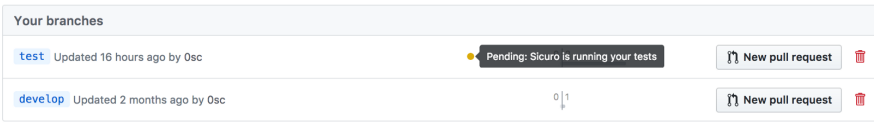
Cuando se registra un webhook, GitHub hace ping a la URL de devolución de llamada subscrita con una carga útil ficticia para cerciorarse de que el punto final dado es alcanzable. Con esto, podemos configurar una ejecución inicial para cada solicitud de suscripción de proyecto.

Al unir esto, el flujo de registro de webhook se convierte en:

> *El usuario hace clic en el enlace de suscripción para un proyecto determinado. Cuando la solicitud llega al servidor back-end, enviamos una solicitud de suscripción a GitHub para el evento PR y commit. Cuando GitHub ejecuta la url de devolución de llamada para confirmar la accesibilidad, activamos un primer trabajo para el proyecto que ejecuta pruebas en la rama maestra.*

Con webhook registrado con éxito, siempre que cualquiera de los eventos de extracción o confirmación tenga lugar en el repositorio del proyecto, GitHub envía la carga del evento a la url de devolución de llamada webhook registrada anteriormente. Y al recibir esta carga útil, la aplicación analiza la información necesaria de la carga útil, crea la carga útil de la solicitud de trabajo adecuada y da la instrucción al servidor de CI para que la construya.

Una de las ventajas de las integraciones como esta es que la mayoría de los VCS permiten comentarios (éxito o fracaso) de la actividad del webhook; que se visualiza correctamente en línea con el cambio para que el usuario pueda ver fácilmente.



Como habrás adivinado, aquí es donde la `updateBuildFunc` función de devolución de llamada mencionada brevemente en la sección del servidor de CI es útil. Aquí hay una implementación de muestra para la función de devolución de llamada:

```
estado  : = & github. RepoStatus {
                TargetURL : github. Cadena ( " nuestra / de
                Contexto : github. Cadena ( " SicuroCI " ),
        }
cliente  : = github. NewClient ( nil )

updateBuildFunc  : =  func ( cadena de estado ) {
   cadena de descripción  var

  estado de cambio {
  caso  " éxito " :
    description = " Tus pruebas pasaron en Sicuro "
  caso  " pendiente " :
    description = " Sicuro está ejecutando tus pruebas "
  caso  " fracaso " :
    description = " Sus pruebas fallaron en Sicuro "
```

Una vez que se inicia la compilación de prueba, la función de devolución de llamada se ejecuta con el estado *en progreso* y luego el veredicto para la compilación, *pasa / falla,* se pasa a la ejecución de la función.

Alojamiento

Ahora que tenemos los elementos básicos del CI juntos, está listo para ser implementado y puesto en uso. En esta sección, analizo cómo implementar la aplicación en AWS, pero cualquiera de los otros proveedores de la nube GCP, Azure, DigitalOcean, etc. también funciona. De hecho, probé con diferentes proveedores antes de establecerme para AWS; *las notas de la prueba crearían otro relato* .

El primer paso para el alojamiento es identificar los diversos recursos necesarios para ejecutar la aplicación y determinar cómo hacer que estén disponibles en la plataforma de su elección. Mirando a través de lo que tenemos hasta ahora, cualquier plataforma requerirá:

- Docker y Docker componen
- Ir tiempo de ejecución

- Claves válidas de GitHub SSH y otras credenciales del proyecto
- Una copia de la aplicación

Vamos a seguir con la configuración de Aws

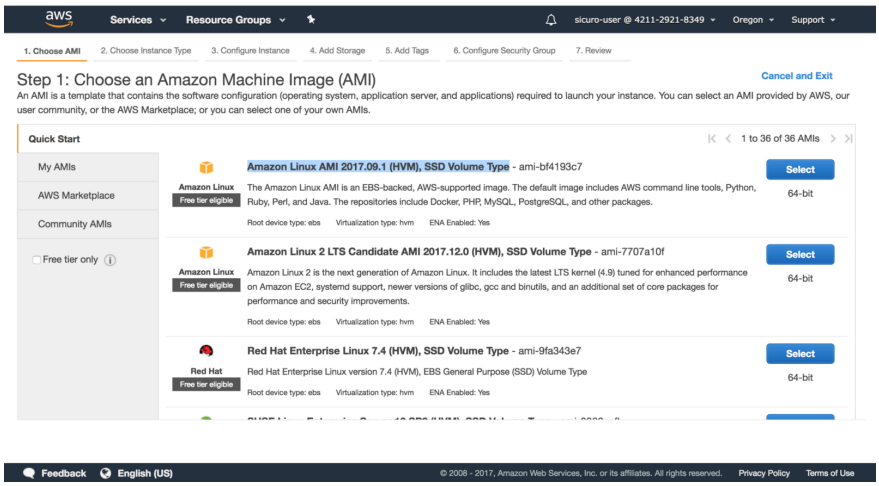
Nota: asumo familiaridad con los principios básicos de Aws; particularmente lanzando instancias EC2, roles AMI, S3. Si le fallo con esta suposición, considere buscar cualquiera de los recursos disponibles sobre el tema. Me parece útil este curso de visión plural .

AMI personalizado

Amazon Machine Image (AMI) es la base desde la que se aprovisionan las instancias de EC2. Ellos " *... contienen la configuración del software (sistema operativo, servidor de aplicaciones y aplicaciones) requerida para iniciar su instancia* ". La elección de AMI depende en gran medida de los requisitos de la aplicación que se implementará en ella.

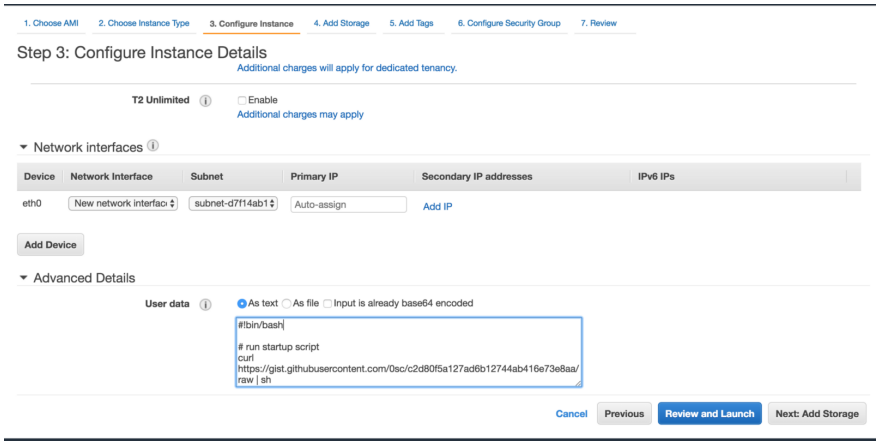
Sicuro, como detallamos anteriormente requiere Docker, Docker compose y un tiempo de ejecución de Golang; y como suele ser el caso de las aplicaciones especializadas, no existe un AMI preconstruido que agrupe todo el requisito. Así que tenemos que lanzar el nuestro.

Primero, generamos una instancia genérica de EC2 ejecutando una imagen base que tiene la dependencia del proyecto requerida. Los AMI personalizados como cabría esperar (y muy parecidos a las imágenes de Docker) se basan en imágenes existentes y en nuestro caso aquí **basaremos el AMI 2017.09.1 (HVM) de Amazon Linux, tipo de volumen SSD** (en gran parte porque es elegible para el Aws nivel libre;))



Luego, en la sección de configurar la instancia, en Detalles avanzados, configuraremos el siguiente comando para que se ejecute

```
#! bin / sh

curl
https://gist.githubusercontent.com/0sc/71975ba7e9c1a32d37c81
5e2e2806920/raw | sh
```



Esto ejecuta la siguiente secuencia de comandos para instalar `docker` , `docker-compose` y `golang` en la imagen

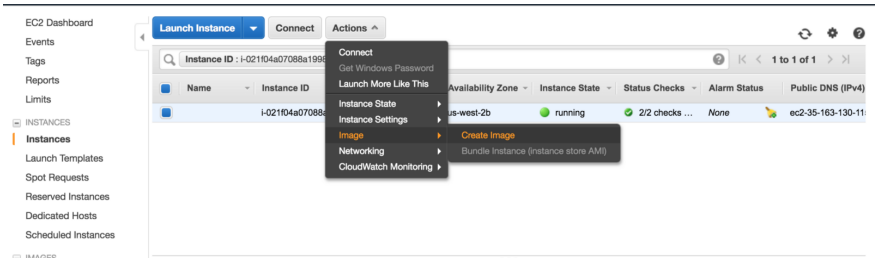
```
#! bin / sh

# Actualizar paquetes instalados y caché de paquetes
sudo yum update -y

# asegúrese de que en la carpeta de inicio
cd  ~ /

# Instalación de Golang

# especificar ir versión para instalar
VERSIÓN = go1.9.linux-amd64.tar.gz

# descargar el archivo go para la versión especificada
sudo curl -O https://storage.googleapis.com/golang/ $ VERSI

# extraer el archivo descargado en la carpeta / usr / local
sudo tar -C / usr / local -xzf $ VERSIÓN

# configurar el espacio de trabajo predeterminado de GOPATH
mkdir -p ~ / go / bin

# establecer las variables env necesarias
cat > ./go-env.sh << EOL
exportar GOPATH = ~ / go
PATH de exportación = $ PATH: / usr / local / go / bin: ~ /
EOL
chmod + x ./go-env.sh
sudo mv ./go-env.sh /etc/profile.d/

# install git para obtener el comando get
# Instalación de Docker
# http://docs.aws.amazon.com/AmazonECS/latest/developerguid
```

Con ese conjunto, podemos continuar y lanzar la imagen.

Una vez que la instancia se está ejecutando, SSH en ella y confirmar que `docker` , `docker-compose` y `golang` si están instalados

```
Last login: Wed Dec 27 13:33:44 2017 from 41.217.115.207

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2017.09-release-notes/
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[ec2-user@ip-10-0-0-147 ~]$ docker version
Client:
 Version:      17.06.2-ce
 API version:  1.30
 Go version:   go1.8.4
 Git commit:   3dfb8343b139d6342acfd9975d7f1068b5b1c3d3
 Built:        Fri Nov 10 00:50:37 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.06.2-ce
 API version:  1.30 (minimum version 1.12)
 Go version:   go1.8.4
 Git commit:   402dd4a/17.06.2-ce
 Built:        Fri Nov 10 00:51:08 2017
 OS/Arch:      linux/amd64
 Experimental: false
[ec2-user@ip-10-0-0-147 ~]$ docker-compose version
docker-compose version 1.16.1, build 6d1ac21
docker-py version: 2.5.1
CPython version: 2.7.13
OpenSSL version: OpenSSL 1.0.1t  3 May 2016
[ec2-user@ip-10-0-0-147 ~]$ go version
go version go1.9 linux/amd64
[ec2-user@ip-10-0-0-147 ~]$
```

Ahora todo está configurado para que creemos una imagen desde la instancia en ejecución. Seleccione la instancia en ejecución desde el panel de instancia de EC2 en su consola de Aws; luego seleccione la *opción Acciones> imagen> crear imagen* . Dale a tu imagen un nombre interesante y crea.



FYI también puede usar la imagen creada para cualquier proyecto que requiera dependencias similares, docker, docker compose :)

Manejo de credenciales con S3

Como todo buen proyecto, debemos ser cautelosos sobre cómo manejamos las credenciales y los secretos del proyecto. Los secretos requeridos por Sicuro incluyen

- Claves GitHub SSH
- Client ID y Client Secret para la aplicación GitHub OAuth

- Webhook Secret para registrar webhooks en proyectos de GitHub
- Secretos de sesión para manejar sesiones de usuario para la aplicación web

Hay un par de maneras discretas de hacer que esta información esté disponible para la aplicación en tiempo de ejecución. Un enfoque, que usaríamos, es el enfoque S3 detallado en esta publicación de blog .

Crea un cubo en S3; llámalo decir **sicuro-secretos** . Suba `.env` al cubo la copia del archivo que contiene los créditos de producción.

```
GITHUB_CLIENT_ID = YOUR-GITHUB-CLIENT-ID
GITHUB_CLIENT_SECRET = YOUR-GITHUB-CLIENT-SECRET
GITHUB_WEBHOOK_SECRET = YOUR-GITHUB-WEBHOOK-SECRET
SESSION_SECRET = ALGUNA
LARGO-RANDOM-STRING-FOR-CODIFICATION-USER-SESSION
DOCKER_IMAGE_REPO = xovox
```

Configure (otra) la clave GitHub SSH para su cuenta de GitHub siguiendo el tutorial aquí . El resultado final debe ser dos archivos: `id_rsa` y que `id_rsa.pub` contiene sus claves privadas y públicas, respectivamente. Añádalos a una `.ssh` carpeta y cárguelos a una `keys` subcarpeta en el depósito S3. Así es como debería verse la estructura final del archivo:

```
sicuro-secrets
  | - keys
       | - .ssh
             | - id_rsa
             | - id_rsa.pub
  | - .env
```

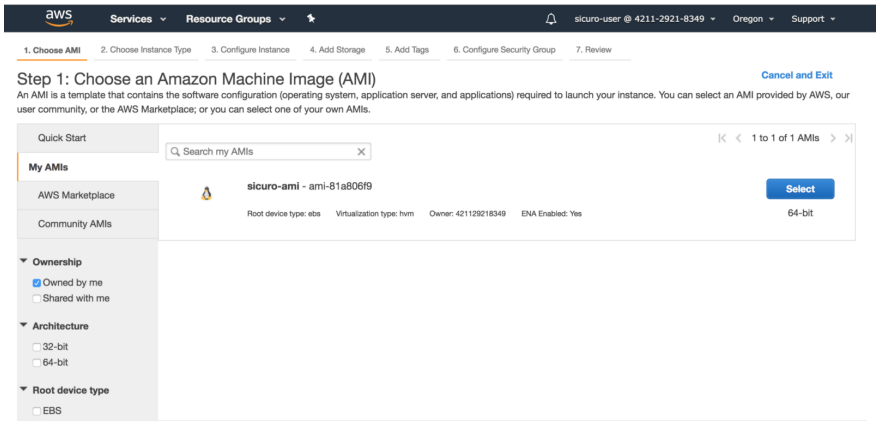
Tenga en cuenta que no hay restricciones en la estructura o el nombre del archivo. La sugerencia aquí se basa en la suposición para el guión proporcionado en la siguiente sección a continuación.

Finalmente, cree un rol de IAM con `read` acceso al segmento S3. Asignaremos esto a instancias de EC2 que ejecuten la aplicación para que puedan extraer contenidos del depósito.

Ahora todo está configurado para ejecutar la aplicación en EC2.

Implementando la aplicación

En el panel de EC2, haga clic para crear una nueva instancia. Cuando se le solicite que use AMI, haga clic para seleccionar de las `MY AMIs` secciones y seleccione la imagen personalizada creada en la sección *AMI personalizada* anterior.



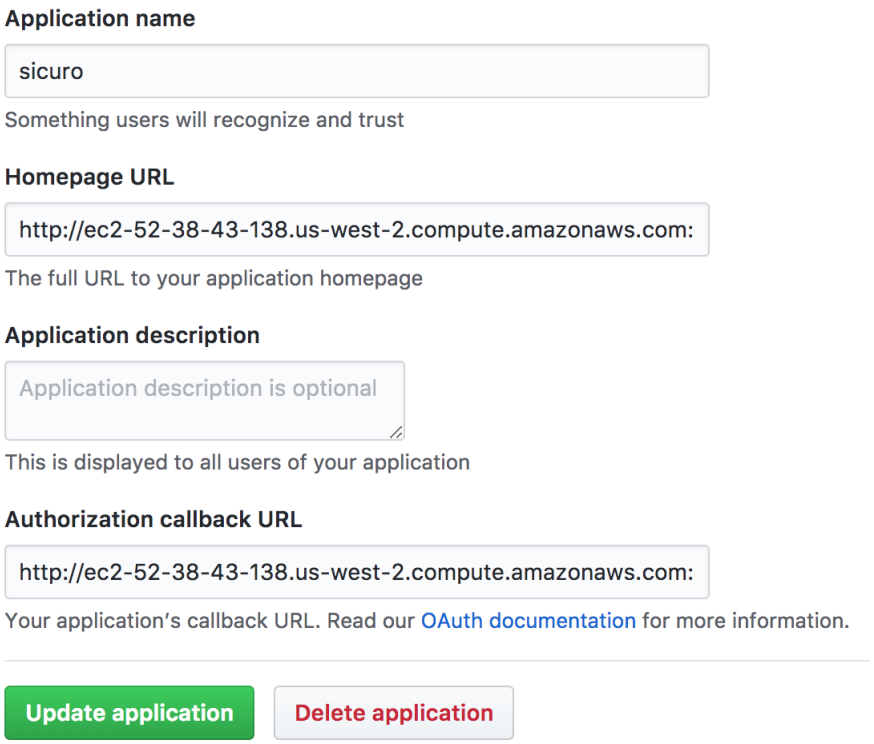
En la configuración de la instancia, asígnele la función IAM creada en la sección anterior y en la sección de detalles avanzados, agregue la siguiente secuencia de comandos de inicio (en el campo de datos del usuario como texto):

```
#! bin / bash

curl
https://gist.githubusercontent.com/0sc/c2d80f5a127ad6b12744a
b416e73e8aa/raw | sh
```

Esto ejecuta el siguiente script de inicio

```
#! bin / bash
exportar GOPATH = ~ / go
PATH de exportación = $ PATH : $ GOPATH / bin: / usr / loca

SICURO_SRC = github.com / 0sc / sicuro
SECRETS_BUCKET_NAME = sicuro-secrets

# buscar el código sicuro
ve a obtener $ SICURO_SRC
cd  $ GOPATH / src / $ SICURO_SRC

# set predefine env vars
eval  $ ( aws s3 cp s3: // $ {SECRETS_BUCKET_NAME} /.env -
# copiar el contenido de la carpeta de claves en la carpeta
aws s3 cp s3: // $ {SECRETS_BUCKET_NAME} / keys / ./ci/ --r

# confirmar env vars se establecen
para  i  en GITHUB_CLIENT_ID GITHUB_CLIENT_SECRET GITHUB_WE
    si [ -z  $ { ! i} ] ;  entonces
        echo  > & 2  " error: missing $ i environment varia
    fi
hecho

```

Copie su DNS público y configúrelo como la URL de devolución de llamada para el inicio de sesión de GitHub OAuth en la sección *Interfaz de usuario* , por ejemplo `http://ec2-52-43-89-100.us-west-2.compute.amazonaws.com:8080/gh/callback`



Su CI está lista; visita la url y pruébalo

- Inicia sesión con Github
- Suscribe uno de tus proyectos
- Ver los registros cuando el CI ejecuta su prueba
- Cree un PR o introduzca nuevos cambios en el proyecto y observe cómo el CI extrae sus cambios, activa una nueva compilación y ejecuta sus pruebas

Resumen

Es interesante las diversas partes móviles de un servidor de CI y cómo colaboran para verificar los cambios de código. En este artículo he explorado cada una de estas partes y cómo volver a implementarlas. La implementación de muestra utilizada para este artículo está disponible en Github con instrucciones sobre cómo ejecutarla localmente, compruébalo aquí .

Gracias por leer y asegúrese de compartir sus ideas en la sección de comentarios.