TTPS opción Ruby

Práctica 4

En esta práctica veremos ejercicios para comprender el funcionamiento de las gemas en Ruby, el uso de Bundler y comenzaremos a utilizar la herramienta para desarrollo de aplicaciones web en Ruby Sinatra.

Gemas y Bundler

1. ¿Qué es una gema? ¿Para qué sirve? ¿Qué estructura tienen?
2. ¿Cuáles son las principales diferencias entre el comando `gem` y Bundler? ¿Hacen lo mismo?
3. ¿Dónde instalan las gemas los comandos `gem` y `bundle` ? > Tip: `gem which` y `bundle show` .
4. ¿Para qué sirve el comando `gem server` ? ¿Qué información podés obtener al usarlo?
5. Investigá un poco sobre *Semantic Versioning* (o *SemVer*). ¿Qué finalidad tiene? ¿Cómo se compone una versión? ¿Ante qué situaciones debería cambiarse cada una de sus partes?
6. Creá un proyecto para probar el uso de Bundler:

    1. Inicializá un proyecto nuevo en un directorio vacío con el comando `bundle init`
    2. Modificá el archivo `Gemfile` del proyecto y agregá la gema `colorputs`
    3. Creá el archivo `prueba.rb` y agregale el siguiente contenido:

    ```
    require 'colorputs'
    puts "Hola!", :rainbow_bl
    ```

    4. Ejecutá el archivo anterior de las siguientes maneras:

    5. `ruby prueba.rb`

    6. `bundle exec ruby prueba.rb`

    7. Ahora utilizá el comando `bundle install` para instalar las dependencias del proyecto

    8. Volvé a ejecutar el archivo de las dos maneras enunciadas en el paso **4**.

    9. Creá un nuevo archivo `prueba_dos.rb` con el siguiente contenido:

    ```
    Bundler.require
    puts "Chau!", :red
    ```

10. Ahora ejecutá este nuevo archivo:

    - `ruby prueba_dos.rb`
    - `bundle exec ruby prueba_dos.rb`

7. Utilizando el proyecto creado en el punto anterior como referencia, contestá las siguientes preguntas:

    1. ¿Qué finalidad tiene el archivo `Gemfile`?
    2. ¿Para qué sirve la directiva `source` del `Gemfile`? ¿Cuántas pueden haber en un mismo archivo?
    3. Acorde a cómo agregaste la gema `colorputs`, ¿qué versión se instaló de la misma? Si mañana se publicara la versión `7.3.2`, ¿esta se instalaría en tu proyecto? ¿Por qué? ¿Cómo podrías limitar esto y hacer que sólo se instalen *releases* de la gema en las que no cambie la *versión mayor* de la misma?
    4. ¿Qué ocurrió la primera vez que ejecutaste `prueba.rb`? ¿Por qué?
    5. ¿Qué cambió al ejecutar `bundle install`?
    6. ¿Qué diferencia hay entre `bundle install` y `bundle update`?
    7. ¿Qué ocurrió al ejecutar `prueba_dos.rb` de las distintas formas enunciadas? ¿Por qué? ¿Cómo modificarías el archivo `prueba_dos.rb` para que funcione correctamente?

8. Desarrollá una gema (llamada `MethodCounter`, por ejemplo) que empaquete toda la funcionalidad implementada en el ejercicio 4 de la práctica 2 (el módulo `Countable`).
   La forma de usarla sería algo similar a esto:

```
require 'method_counter'

class MiClase
  include MethodCounter::Countable

  def hola
    puts "Hola"
  end

  def chau
    puts "Chau"
  end

  count_invocations_of :hola, :chau
end
```

Sinatra

1. ¿Qué es Rack? ¿Qué define? ¿Qué requisitos impone?
2. Implementá una *app* llamada “MoL” de Rack que responda con un número al azar entre `1` y `42`,

y que devuelva el *status* HTTP `200` sólo en caso que el número a devolver sea `42`, en cualquier otro caso debe retornar un *status* `404`.

3. Sinatra se define como "*DSL para crear aplicaciones web*". ¿Qué quiere decir esto? ¿Qué es un *DSL*?
4. Implementá la misma *app* "MoL" de antes, ahora utilizando Sinatra para obtener el mismo resultado.
5. Utilizando Sinatra, desarrollá una aplicación web que tenga los siguientes *endpoints*:
   - `GET /` lista todos los endpoints disponibles (sirve a modo de documentación)
   - `GET /mcm/:a/:b` calcula y presenta el mínimo común múltiplo de los valores numéricos `:a` y `:b`
   - `GET /mcd/:a/:b` calcula y presenta el máximo común divisor de los valores numéricos `:a` y `:b`
   - `GET /sum/*` calcula la sumatoria de todos los valores numéricos recibidos como parámetro en el *splat*
   - `GET /even/*` presenta la cantidad de números pares que hay entre los valores numéricos recibidos como parámetro en el *splat*
   - `POST /random` presenta un número al azar
   - `POST /random/:lower/:upper` presenta un número al azar entre `:lower` y `:upper` (dos valores numéricos)
6. Implementá un *middleware* para Sinatra que modifique la respuesta del web server y "tache" cualquier número que aparezca en el *body* de la respuesta, cambiando cada dígito por un caracter `X`. Utilizalo en la aplicación anterior para corroborar su funcionamiento.
7. Implementá otro *middleware* para Sinatra que agregue una cabecera a la respuesta HTTP, llamada `X-Xs-Count`, cuyo valor sea la cantidad de caracteres `X` que encuentra en el *body* de la respuesta. ¿Cómo debés incluirlo en tu *app* Sinatra para que este *middleware* se ejecute **antes** que el desarrollado en el punto anterior?
8. Desarrollá una aplicación Sinatra para jugar al ahorcado. La aplicación internamente debe manejar una lista de palabras (cada una asociada a algún identificador de tu elección y a información sobre los intentos realizados para adivinar esa palabra), donde cada una representa una partida de ahorcado que puede ser jugada una sóla vez por ejecución del servidor de la aplicación web.
   La misma debe poseer las siguientes URLs:
   - `POST /` inicia una partida. Internamente tomará una palabra al azar de entre las de la lista, y luego debe redirigir (con un *redirect* HTTP) a la URL propia de la partida (utilizando el identificador de la palabra elegida) para que el jugador pueda comenzar a adivinar.
   - `GET /partida/:id` muestra el estado actual de la partida (letras adivinadas, intentos fallidos y cantidad de intentos restantes). Si se adivinó la palabra o no quedan más intentos, deberá reflejarse también en la
   - `PUT /partida/:id` acepta un parámetro por `PUT` llamado `intento` que debe contener la letra que el jugador intenta para adivinar la palabra. Internamente la aplicación chequeará si se pueden hacer más intentos en la partida, en caso afirmativo actualizará el estado de la partida, y en respuesta deberá hacer un *redirect* HTTP a la partida (a `/partida/:id`) para mostrar al jugador el estado de su partida.

Referencias

A la hora de aprender un nuevo lenguaje, una herramienta o un framework, es fundamental que te familiarices con sus APIs. Sea conocer clases base del lenguaje o parte de la herramienta que estés comenzado a utilizar, las APIs que te provea serán la forma de sacarle provecho. Si además disponés de guías o *tutoriales* para complementar y guiarte en el aprendizaje (como ocurre en el caso de Sinatra y Rails), ¡mejor aún!

Por eso, te dejamos en esta sección los links para que puedas consultar la documentación de las herramientas que ves en esta práctica:

- Rubygems - https://rubygems.org
  - Guías
- Bundler - http://bundler.io
  - Motivación y breve ejemplo
  - Gemfile
- Sinatra - http://sinatrarb.com/
  - APIs
  - Guía rápida en inglés y en español
  - Índice de documentación
  - Presentación en español