Part 2: RESTful thinking for Wordguesser

Note: Part 2 is just reading/background info for Part 3.

Goals: Understand how to expose your app's behaviors as RESTful actions in a SaaS environment, and how to preserve game state across (stateless) HTTP requests to your app using the appserver's provided abstraction for cookies.

What you will do: Create a Sinatra app that makes use of the WordGuesserGame logic developed in the previous part, allowing you to play Wordguesser via a browser.

Game State

Unlike a shrinkwrapped app, SaaS runs over the stateless HTTP protocol, with each HTTP request causing something to happen in the app. And because HTTP is stateless, we must think carefully about the following 2 questions:

1. What is the total state needed in order for the next HTTP request to pick up the game where the previous request left off?
2. What are the different game actions, and how should HTTP requests map to those actions?

The widely-used mechanism for maintaining state between a browser and a SaaS server is a **cookie**. The server can put whatever it wants in the cookie (up to a length limit of 4K bytes); the browser promises to send the cookie back to the server on each subsequent request. Since each user's browser gets its own cookie, the cookie can effectively be used to maintain per-user state.

In most SaaS apps, the amount of information associated with a user's session is too large to fit into the 4KB allowed in a cookie, so as we'll see in Rails, the cookie information is more often used to hold a pointer to state that lives in a database. But for this simple example, the game state is small enough that we can keep it directly in the session cookie.

Self Check Question

▶ Enumerate the minimal game state that must be maintained during a game of Wordguesser.

The game as a RESTful resource

Self Check Question

▶ Enumerate the player actions that could cause changes in game state.

In a service-oriented architecture, we do not expose internal state directly; instead we expose a set of HTTP requests that either display or perform some operation on a hypothetical underlying **resource**. **The trickiest and most important part of RESTful design is modeling what your resources are and what operations are possible on them.**

In our case, we can think of the game itself as the underlying resource. Doing so results in some important design decisions about how routes (URLs) will map to actions and about the game code itself.

Since we've already identified the game state and player actions that could change it, it makes sense to define the game itself as a class. An instance of that class is a game, and represents the resource being manipulated by our SaaS app.

Mapping resource routes to HTTP requests

Our initial list of operations on the resource might look like this, where we've also given a suggestive name to each action:

1. `create` : Create a new game
2. `show` : Show the status of the current game
3. `guess` : Guess a letter

Self Check Question

▶ For a good RESTful design, which of the resource operations should be handled by HTTP GET and which ones should be handled by HTTP POST?

HTTP is a request-reply protocol and the Web browser is fundamentally a request-reply user interface, so each action by the user must result in something being displayed by the browser. For the "show status of current game" action, it's pretty clear that what we should show is the HTML representation of the current game, as the `word_with_guesses` method of our game class does.

But when the player guesses a letter--whether the guess is correct or not--what should be the "HTML representation" of the result of that action?

Answering this question is where the design of many Web apps falters.

In terms of game play, what probably makes most sense is after the player submits a guess, display the new game state resulting from the guess. **But we already have a RESTful action for displaying the game state.** So we can plan to use an **HTTP redirect** to make use of that action.

This is an important distinction, because an HTTP redirect triggers an entirely new HTTP request. Since that new request does not "know" what letter was guessed, all of the responsibility for **changing** the game state is associated with the guess-a-letter RESTful action, and all of the responsibility for **displaying** the current state of the game without changing it is associated with the display-status action. This is quite different from a scenario in

which the guess-a-letter action **also** displays the game state, because in that case, the game-display action would have access to what letter was guessed. Good RESTful design will keep these responsibilities separate, so that each RESTful action does exactly one thing.

A similar argument applies to the create-new-game action. The responsibility of creating a new game object rests with that action (no pun intended); but once the new game object is created, we already have an action for displaying the current game state.

So we can start mapping our RESTful actions in terms of HTTP requests as follows, using some simple URIs for the routes:

Route and action	Resource operation	Web result
`GET /show`	show game state	display correct & wrong guesses so far
`POST /guess`	update game state with new guessed letter	redirect to `show`
`POST /create`	create new game	redirect to `show`

In a true service-oriented architecture, we'd be nearly done. But a site experienced through a Web browser is not quite a true service-oriented architecture.

Why? Because a human Web user needs a way to `POST` a form. A `GET` can be accomplished by just typing a URL into the browser's address bar, but a `POST` can only happen when the user submits an HTML form (or, as we'll see later, when AJAX code in JavaScript triggers an HTTP action).

So to start a new game, we actually need to provide the user a way to post the form that will trigger the `POST /create` action. You can think of the resource in question as "the opportunity to create a new game." So we can add another row to our table of routes:

`GET /new`	give human user a chance to start new game	display a form that includes a "start new game" button

Similarly, how does the human user generate the `POST` for guessing a new letter? Since we already have an action for displaying the current game state (`show`), it would be easy to include on that same HTML page a "guess a letter" form that, when submitted, generates the `POST /guess` action.

We will see this pattern mirrored later in Rails: a typical resource (such as the information about a player) will have `create` and `update` operations, but to allow a human being to provide the data used to create or update a player record, we will have to provide `new` and `edit` actions respectively that allow the user to enter the information on an HTML form.

Self Check Questions

▶ Why is it appropriate for the `new` action to use `GET` rather than `POST` ?

▶ Explain why the `GET /new` action wouldn't be needed if your Wordguesser game was called as a service in a true service-oriented architecture.

Lastly, when the game is over (whether win or lose), we shouldn't be accepting any more guesses. Since we're planning for our `show` page to include a letter-guess form, perhaps we should have a different type of `show` action when the game has ended---one that does **not** include a way for the player to guess a letter, but (perhaps) does include a button to start a new game. We can even have separate pages for winning and losing, both of which give the player the chance to start a new game. Since the `show` action can certainly tell if the game is over, it can conditionally redirect to the `win` or `lose` action when called.

The routes for each of the RESTful actions in the game, based on the description of what the route should do:

Show game state, allow player to enter guess; may redirect to Win or Lose	GET /show
Display form that can generate `POST /create`	GET /new
Start new game; redirects to Show Game after changing state	POST /create
Process guess; redirects to Show Game after changing state	POST /guess
Show "you win" page with button to start new game	GET /win
Show "you lose" page with button to start new game	GET /lose

Summary of the design

You may be itchy about not writing any code yet, but you have finished the most difficult and important task: defining the application's basic resources and how the RESTful routes will map them to actions in a SaaS app. To summarize:

- We already have a class to encapsulate the game itself, with instance variables that capture the game's essential state and instance methods that operate on it when the player makes guesses. In the model-view-controller (MVC) paradigm, this is our model.
- Using Sinatra, we will expose operations on the model via RESTful HTTP requests. In MVC, this is our controller.
- We will create HTML views and forms to represent the game state, to allow submitting a guess, to allow starting a new game, and to display a message when the player wins or loses. In MVC, these are our views.

Note that Sinatra does not really enforce MVC or any other design pattern---if anything, it's closest to the Page Controller pattern, where we explicitly match up each RESTful request with an HTML view---but it's a simple enough framework that we can use it to implement MVC in this app since we have only one model. As we'll see later, more powerful MVC-focused frameworks like Rails are much more productive for creating apps that have many types of models.

Next: Part 3 - Connecting Wordguesser to Sinatra