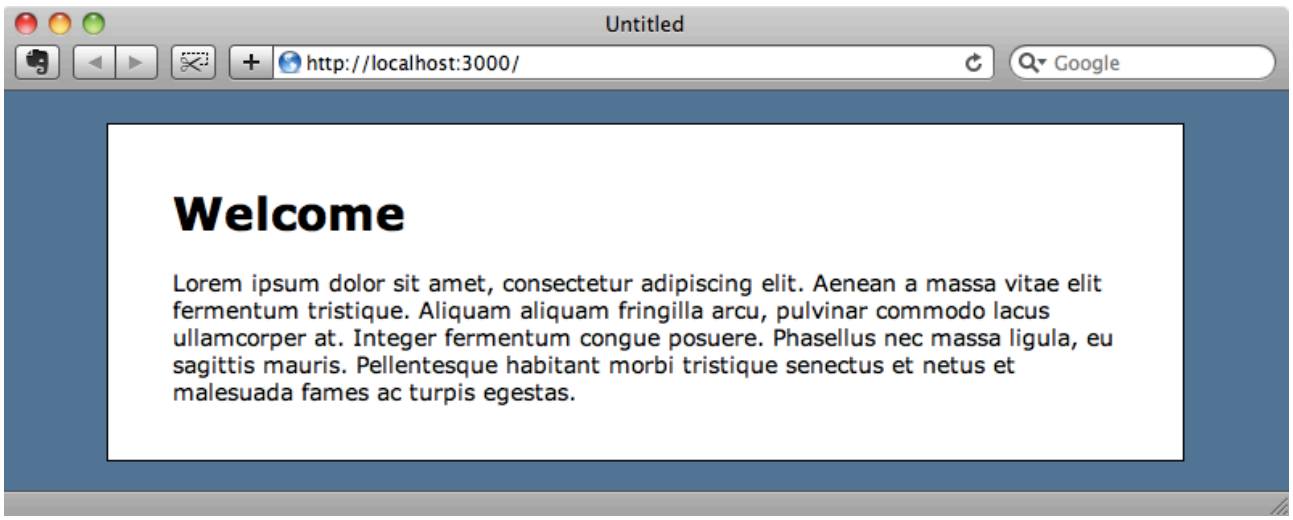

# Episode 256

# I18n Backends

The topic of internationalization was last covered in episode 138 [watch[1], read[2]]. By default Rails stores internationalization information in YAML files, but in this episode we'll show you how use a different backend.

Below is a page from a very simple Rails site on which we want to internationalize the header text.




It's easy to internationalize a piece of text in a Rails application by using the translate method, `t`, but we then need edit a YAML file for each language that our application supports and add the key and the text for that language. For large applications this can quickly become tedious and it isn't really the developer's job to write all of the internationalized text anyway. It would be much better to provide a web interface that allows the app's administrators to add and edit the internationalized text.

Thankfully the internationalization gem supports several different backends which means that we're not restricted to using YAML, we can choose to use any database backend we want. By default Rails uses a Simple[3] backend which handles loading the YAML files and managing the translations through them. The backend we'll focus on in this episode is the Key-Value backend[4] which allows us to use any key-

---

[1] http://railscasts.com/episodes/138-i18n

[2] http://asciicasts.com/episodes/138-i18n

[3] https://github.com/svenfuchs/i18n/blob/master/lib/i18n/backend/simple.rb

[4] https://github.com/svenfuchs/i18n/blob/master/lib/i18n/backend/key_value.rb

value store for managing the translations. There is also an ActiveRecord store which has been extracted out into a separate gem[5]. This works but as the translations on the page are accessed frequently in every page of the application, ActiveRecord isn't really the best approach. We really want the translations in memory rather than in a SQL database. We could use caching to get around this issue but then we need to worry about expiring the cache when the translations are modified. For these reasons a key-value store is the way to go and its this approach that we'll be demonstrating in this episode.

## Changing The Backend

First we'll need to change the static text in the header so that it uses a translation instead. We'll point it to a key called `welcome`.

/app/views/home/index.html.erb

```
<h1><%= t('welcome') %></h1>
```

We can then add the translated text into the English YAML file.

/config/locales/en.yml

```
en:
  welcome: "Welcome"
```

If we refresh the page now we'll see the text from the YAML file displayed.

Now that we have some translated text using the default back end let's look at changing it. At the top of the `key_value.rb` file are some comments explaining how it should be used, including an example of how to implement an alternative backend.

```
#   I18n.backend = I18n::Backend::KeyValue.new ↵
      (Rufus::Tokyo::Cabinet.new('*'))
```

We need to create a new `I18n::Backend::KeyValue` and pass in the key-value store that we want to use. This will then be used as the backend for the translations. The store needs to be able to respond to three methods, one to get a key's value, one

[5] https://github.com/svenfuchs/i18n-active_record

to set it and one to list all of the keys. These are shown in the comments. Most key-value stores in Ruby support these methods and so we can use one them out of the box.

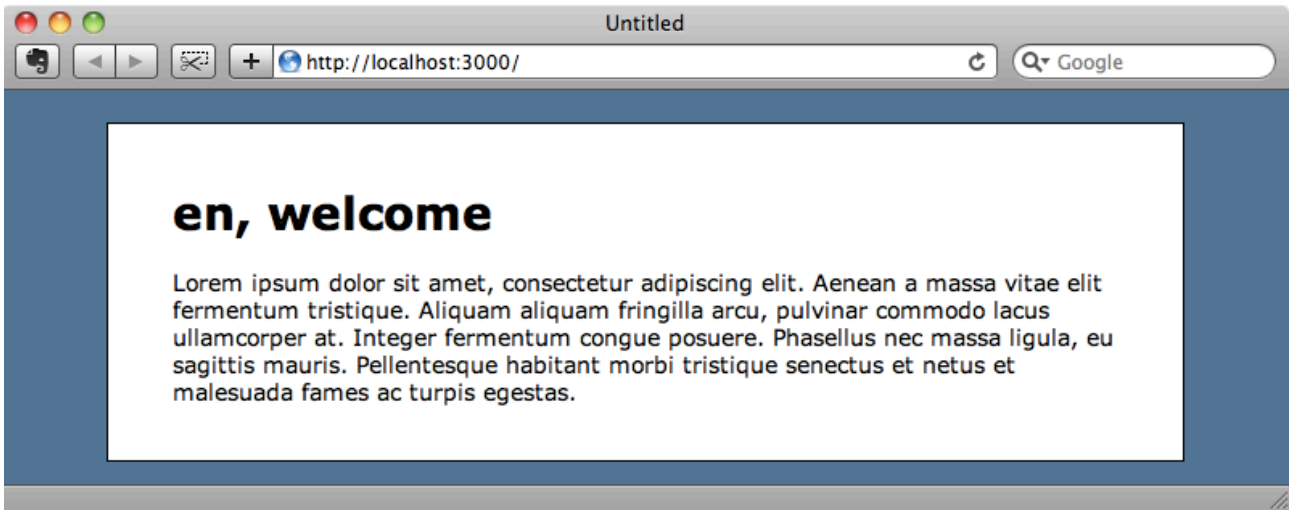
```
# * store#[](key)          - Used to get a value
# * store#[]=(key, value) - Used to set a value
# * store#keys             - Used to get all keys
```

We have enough information now to start moving our application's internationalization backend so let's begin. First we'll create a new file in the `/config/initializers` directory called `i18n_backend.rb`.

`/config/initalizers/i18n_backend.rb`

```
I18n.backend = I18n::Backend::KeyValue.new({})
```

The example in the comments uses a Tokyo Cabinet backend but to get the application up and running we've used an empty hash. Obviously we wouldn't do this in a real application but as it's the simplest thing that supports the three methods listed above it'll do for demonstration purposes. If we restart our application now and go to the home page we'll see that the title doesn't look right any more.




If we look at the source code we'll see that we now have a missing translation.

```
<h1><span class="translation_missing">en, welcome</span></h1>
```

As the application is now using its new backend, the translation is marked as missing. Even though it's still there in the YAML file, it's not in our new backend. We'll create the web interface now to enable users to add translations to the new backend. This will all be handled by a new `TranslationsController` with one action, `index`.

```
$ rails g controller translations index
```

We want this new controller to behave like a resource so we'll replace the generated route (`get "translations/index"`) in the `routes` file with a call to `resources`.

/config/routes.rb

```
Intn::Application.routes.draw do
  resources :translations
  root :to => "home#index"
end
```

In the `index` action we want to display the existing translations and so we'll need to get the translations hash from the new backend store. We can do that by called `I18n.backend.store`.

/app/controllers/translations_controller.rb

```
class TranslationsController < ApplicationController
  def index
    @translations = I18n.backend.store
  end
end
```

In the view we can then iterate through the translations and display them.

/app/views/translations/index.html.erb

```
<h1>Translations</h1>

<ul>
  <% @translations.each do |key, value| %>
    <li><%= key %>: <%= value %></li>
  <% end %>
</ul>
```

This code loops through all of the translations in the hash and displays each key and value in a list. We'll need a way to add new translations so we'll add a form below the list.

/app/views/translations/index.html.erb

```
<h1>Translations</h1>

<ul>
  <% @translations.each do |key, value| %>
    <li><%= key %>: <%= value %></li>
  <% end %>
</ul>

<h2>Add Translation</h2>
<%= form_tag translations_path do %>
 <p>
   <%= label_tag :locale %><br />
   <%= text_field_tag :locale %>
  </p>
  <p>
    <%= label_tag :key %><br />
    <%= text_field_tag :key %>
  </p>
  <p>
    <%= label_tag :value %><br />
    <%= text_field_tag :value %>
  </p>
  <p><%= submit_tag "Submit" %></p>
<% end %>
```

This form will POST to the `TranslationController`'s `create` action. The form has three fields: one for the translation's locale, such as `en` for English;  one for the key, which is how we'll identify the translation in the view files, and one for the translated text itself.

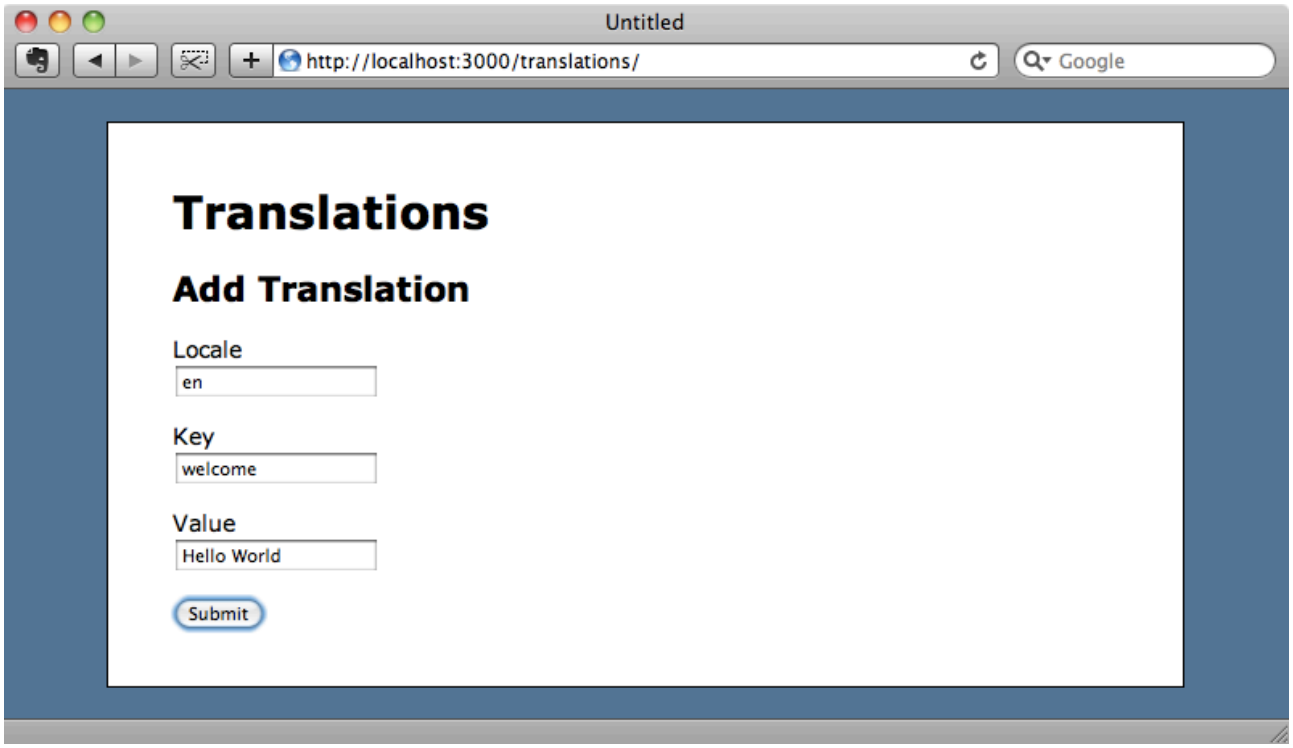
In the `create` action we want to add a new translation based on the supplied form values and we can do that by calling `I18n.backend.store_translations`. This takes three arguments: the first is the locale, which we get from the form and the second is a hash and can be anything we want. We'll pass in the key and value from

the form fields. The final argument, `escape`, determines if the full stops are escaped in the key or not. We'll use full stops to separate different parts of the key so we'll set it to `false`.

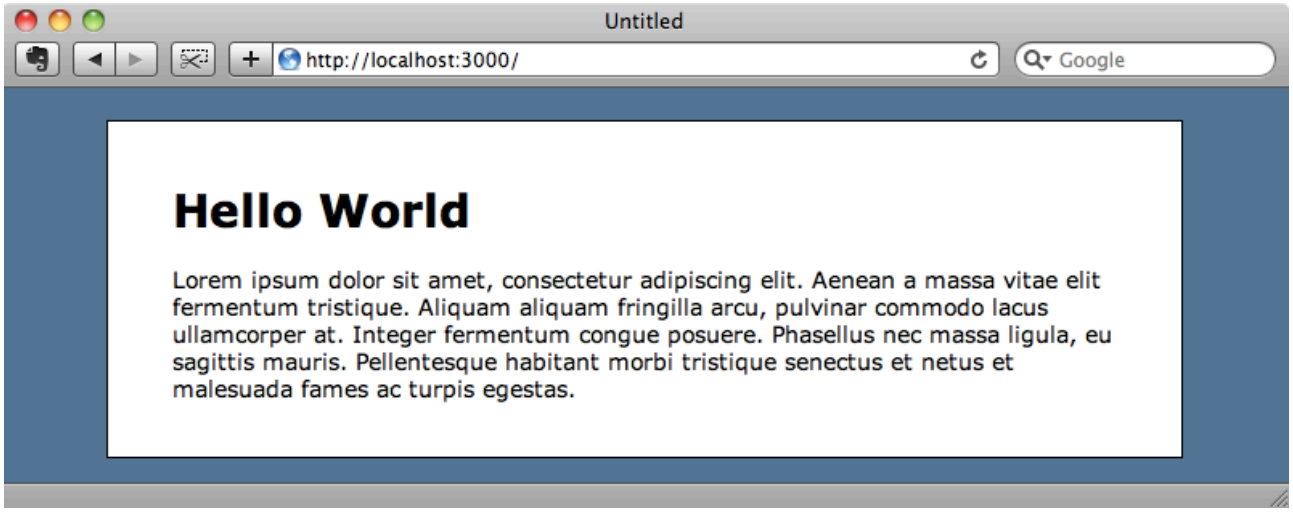
/app/controllers/translations_controller.rb

```
def create
  I18n.backend.store_translations(params[:locale], ↵
    {params[:key] => params[:value]}, :escape => false)
  redirect_to translations_url, :notice => "Added translations"
end
```

We can try the new form out by adding the missing translation from the home page.




When we go back to the home page now we'll see the translated header fetched from our new backend.




## Using Redis For The Backend

Our new backend works well now but as we're storing the values in a Ruby hash they're all lost when we restart the web server. We need a more persistent store for the translations and for this application we're going to use Redis[6] which is a simple persistent key-value store.

If you're using a Mac the easiest way to install Redis is via HomeBrew[7]. To install it this way we need to run

```
$ brew install redis
```

Once its installed follow the instructions to start the Redis server. This is generally as straightforward as running `redis-server`.

We'll have to install the Redis gem to use Redit in our application. This is done by adding a reference to the gem in our `Gemfile` and then running the `bundle` command.

---

[6] http://redis.io/

[7] https://github.com/mxcl/homebrew

/Gemfile

```
source 'http://rubygems.org'

gem 'rails', '3.0.5'
gem 'sqlite3'
gem 'nifty-generators'
gem 'redis'
```

We can now replace the hash in the backend initializer with a Redis database.

/config/initializers/i18n_backend.rb

```
I18n.backend = I18n::Backend::KeyValue.new(Redis.new)
```

That's all that's really necessary, although we might want to use the `:db` option to specify a database and to change the database that's used depending on whether the application's running in development, test or production mode.

As we've changed the key-value store we'll need to change the translations view code, too. Currently it loops through a hash, but it now will be looping through a Redis database so we'll need to change the following piece of code:

/app/views/translations/index.html.erb

```
<ul>
  <% @translations.each do |key, value| %>
    <li><%= key %>: <%= value %></li>
  <% end %>
</ul>
```

to this:

/app/views/translations/index.html.erb

```
<ul>
  <% @translations.keys.each do |key| %>
    <li><%= key %>: <%= @translations[key] %></li>
  <% end %>
</ul>
```

Now `@translations` points to a Redis database instance. which doesn't respond to

each, so we'll need to iterate instead over its keys. In the block we can show the key and the value for that key.

We now have a persistent store for our translations and any that we add through the form will survive when the application's server is restarted.

## Adding a Fallback Backend

We now have our alternative backend in place but what if we'd still like to use the YAML files for some of the translations? We'll show you next how to use the YAML file as a fallback for values that aren't available in the key-value store.

To do this we need to make some changes to the way we define the backend in the initializer file that we created earlier. Instead of using a KeyValue backend directly we'll use a Chain backend. Any backends that we pass in to the Chain backend will be called in order until one of them responds to a given translation key.

/config/initializers/i18n_backend.rb

```
I18n.backend = I18n::Backend::Chain.new ↵
  (I18n::Backend::KeyValue.new(Redis.new), I18n.backend)
```

We pass in our Redis backend first and then the default backend. Our application will now look for translations in the Redis database and, if it fails to find the appropriate key there, will look in the appropriate YAML file.

Initalizing the backend this way makes it more difficult to access the key-value store directly. To get around this we can move our database out into a constant.

/config/initializers/i18n_backend.rb

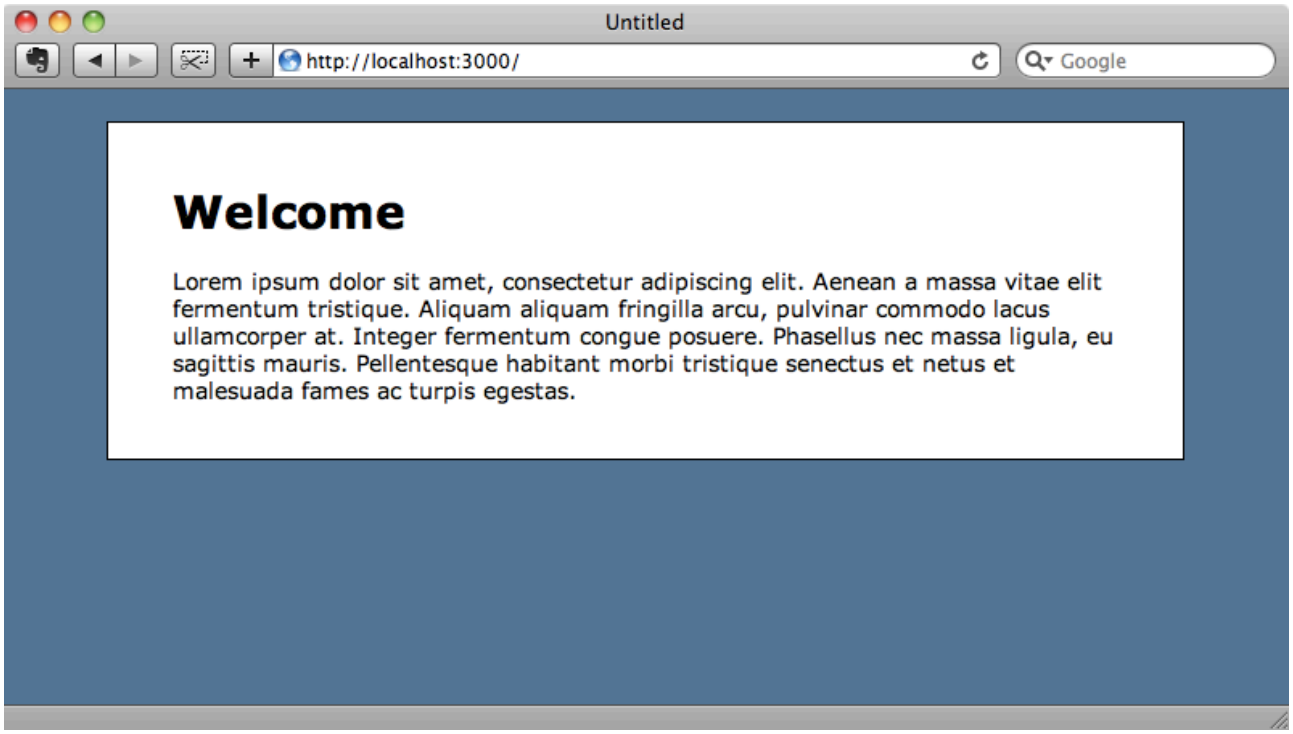
```
TRANSLATION_STORE = Redis.new
I18n.backend = I18n::Backend::Chain.new ↵
  (I18n::Backend::KeyValue.new(TRANSLATION_STORE), I18n.backend)
```

We can now use this constant wherever we need to access the key-value store in our application, such as inside the TranslationsController. We can no longer call I18n.backend.store here.

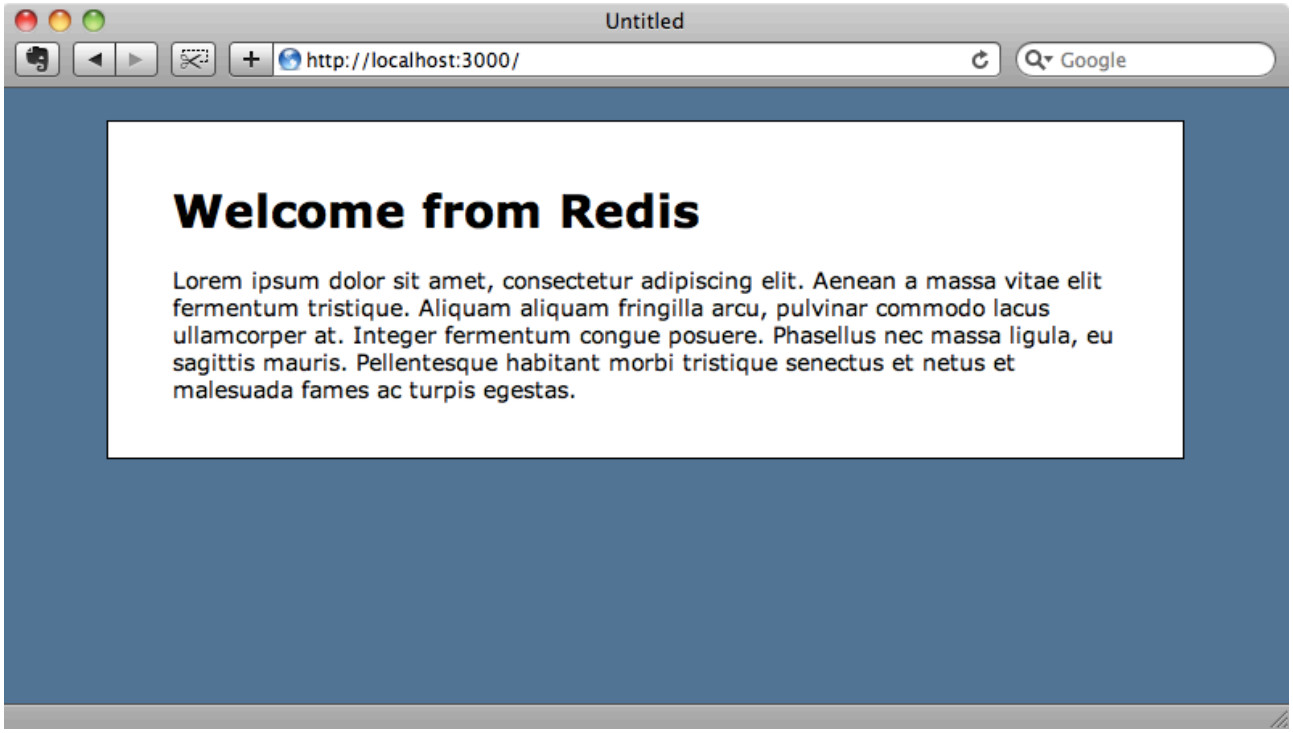
/app/controllers/translations_controller.rb

```
def index
  @translations = TRANSLATION_STORE
end
```

If we remove the stored translations from our Redis store and go to the home page now we'll see the translated text from the YAML file.




When we add the translation back in it will take precedence and we'll see the value from the Redis database.




That's it for this episode. We now have a system in place that lets us edit translations through a web interface rather than by having to manually edit YAML files. If we were to do something like this in a production application there's plenty that could be done to improve the user experience, but the basics are there.

If you want information about this topic then it's well worth taking a look at José Valim's upcoming book "Crafting Rails Applications"[8] which is currently in beta and which was a great help in writing this episode.

[8] http://pragprog.com/titles/jvrails/crafting-rails-applications