Versioned APIs with Phoenix

Elvio Vicosa

About the author

My name is Elvio Vicosa and I am a software developer based in Berlin, Germany. I've been working with software for over 10 years and throughout that time, I had the chance to found my own consultancy company, work in one of the greatest consultancy companies in the world, work as consultant in Fortune 500 companies, lead software development teams in small startups and multi-milion companies, join a health-tech startup as CTO.

Although I have been using Erlang in the latest years, I discovered (and fall in love with) Elixir one year ago. Since then I am using it whenever it makes sense.

You can find more information about me and also my writtings about Elixir and Phoenix at **http://www.elviovicosa.com**.

Elvio

About this book

Are you developing (or planning to develop) an API using Phoenix? If so, this book was written for you.

“Versioned APIs with Phoenix” is about different ways of implementing API versioning and how they can easily be done in Phoenix.

This e-book is free. You can find the code samples at https://github.com/elvio/versioned-apis-with-phoenix. It includes the project code and tests.

If you find any typo, please send an email to elvio@elviovicosa.com.

Introduction

To build a house, it’s necessary to go through different phases before stacking the first brick. The architect needs to pay attention to everything, drawing all the technical details and documenting the architecture design.

After the drawings are finished, the construction materials need to be chosen and things like the brick type, the insulation layer, the heating and plumbing systems affect how the house is going to be built.

The construction phase starts when the architect gathered all the information and after some time the house is (hopefully) finished.

What would happen, if after the house is finished, the family that owns the house asks the architect to change the brick type? What would happen if the family realises that they want an extra floor?

On the traditional manufacturing and construction industries, changes like these are not common, because they have prohibitive costs, and usually they are impossible.

Software development is often compared with the construction of a house, where a development team would be able to forecast and plan all the details involving a project, but it couldn’t be more different. Differently from building a house, software evolves.

Doesn’t matter the industry you are in or the programming language you use. If you are developing software, one thing is guaranteed: it will change.

Product Evolution

Imagine you are a member of a software development team. Your company is building a mobile application that allows users to add information to their profiles. The application was not released yet. You work in the backend team, responsible for building and deploying the API that will support this mobile application.

That's one of the user stories you are going to work on:

```
Scenario: User can set her/his favorite sport
---------------------------------------------
Given I am a User
And I am on my profile screen
When I select one favorite sport from the sports list
Then the item should be shown in my public profile
```

Based on this user story, the backend team decides that storing the favorite sport item as a string in the database is a good approach and communicate that decision to the mobile development team, that will be consuming the API from the iOS and Android applications.

After a couple of weeks the company releases the application and it instantly becomes a huge success. It is downloaded by millions of people and everyone is happy.

User feedback starts popping up and the product team recognizes that one feature is really annoying the users: they are only able to select one favorite sport, but some of them want to select two or more. The product team shares a user feedback:

"Hey. Your app is amazing and I am using it everyday. I just want to let you know that the favorite sport thing is really weird. I love skiing during winter and sailing during summer, so for me it's impossible to choose only one. Do you plan supporting multiple favorite sports?"

With that in mind, a new user story is prioritized for the next sprint:

```
Scenario: User can set multiple favorite sports
-----------------------------------------------
Given I am a User
And I am on my profile screen
When I select multiple favorite sports from the sports list
Then these items should be shown in my public profile
```

You and the backend team decide that the best approach is to use an Array, migrating all the existing "favorite_sport" strings to an Array of one element, allowing the users of newer versions to define multiple favorite sports.

There is one problem though: If you release a new version of the application where the API response is an Array of favorite sports, instead of a String of a single favorite sport, the users using the old version of the application won’t be able to show nor set the “favorite sport” item, because the API no longer supports a single “favorite sport”.

To not cause damage to the company, the development team needs to support both the previous version of the application (used by people who didn’t update yet) and also the new one.

API versioning

Versioning enables an API to response with different content, based on the information sent by the client.

As we know, software evolves and sometimes is really hard to keep its backward compatibility. As seen in the previous example, it’s very common to get into situations where the product evolves towards an incompatible version.

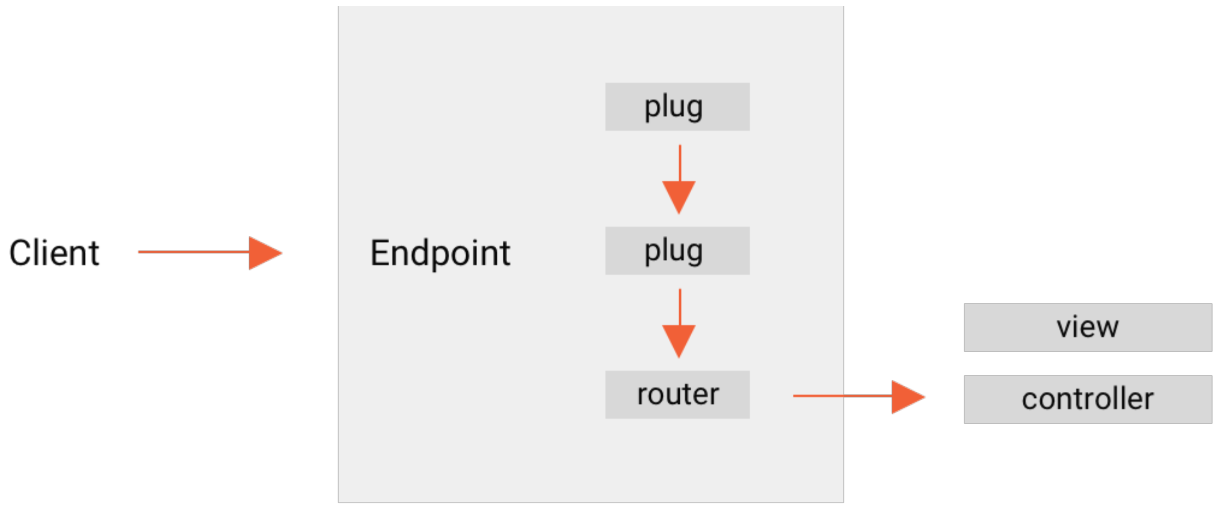
Versioning your API solves this kind of problem, where a team needs to support variant versions of a product at the same time, without having to drop support to old versions, nor compromise future development.

There are different strategies on how to implement a version-aware API and throughout this book, we are going to cover three different strategies:

- Versioning using a request parameter
- Versioning using the URL
- Versioning using the Accept header

A Phoenix request

When a client performs a request to a Phoenix application, that's what usually happens:



It starts with the endpoint of the application, that pipes the connection through a number of functions (plugs).

Every plug is capable of performing transformations in the connection struct, passing the changed connection to the next plug down the line.

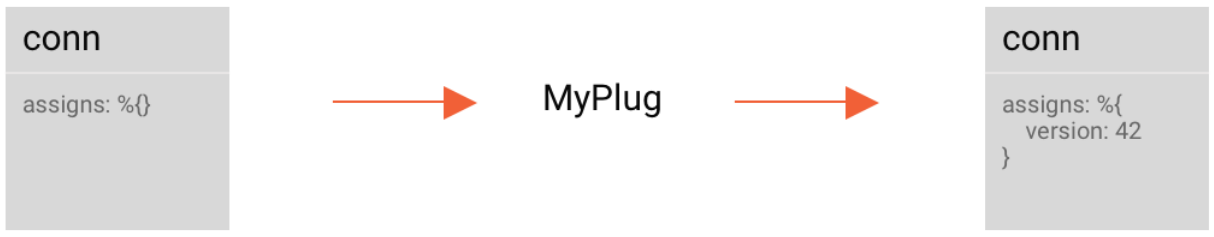
Let's take a look at a plug example:

```
defmodule MyPlug do
  import Plug.Conn

  def init(opts), do: opts

  def call(conn, _opts) do
    assign(conn, :number, 42)
  end
end
```

The plug `MyPlug` sets the value of `:number` to `42` inside the connection assigns. You can imagine it to something similar to this:



The router plug is the last thing in the endpoint. Inside the router, you will find pipelines. A pipeline is basically a group of plugs, that receives a conn struct, pipes it through the group, returning a conn struct.

After passing through the router pipelines, the connection is then dispatched to a controller, where a response will be rendered.

Common code

In this book, you will see three different strategies to version our API. Independently of the strategy, our changes will only affect the routing layer.

Bellow you will find the code of the controller and view that we will use throughout the book. These two files won't change and it's important to understand a bit about them before we continue.

Controller

```
 1 defmodule Profiles.UserController do
 2   use Profiles.Web, :controller
 3   alias Profiles.User
 4 
 5   def show(%{assigns: %{version: :v1}}=conn, _params) do
 6     user = User.build(:v1)
 7     render conn, "show.v1.json", user: user
 8   end
 9   def show(%{assigns: %{version: :v2}}=conn, _params) do
10     user = User.build(:v2)
11     render conn, "show.v2.json", user: user
12   end
13 end
```

web/controllers/user_controller.ex - Github

The `UserController` is able to handle two versions of the API:`v1` and `v2`. You can see that we use pattern-matching to distinguish between the versions.

On line `5` we pattern-match the version `:v1`, rendering the `show.v1.json` view on line `7`.

Something quite similar happens on line `9`, this time we pattern-match the version `:v2`, rendering the `show.v2.json` view on line `11`.

View

```elixir
 1 defmodule Profiles.UserView do
 2   use Profiles.Web, :view
 3
 4   def render("show.v1.json", %{user: user}) do
 5     %{
 6       id: user.id,
 7       name: user.name,
 8       favorite_sport: Enum.at(user.favorite_sports, 0)
 9     }
10   end
11   def render("show.v2.json", %{user: user}) do
12     %{
13       id: user.id,
14       name: user.name,
15       favorite_sports: user.favorite_sports
16     }
17   end
18 end
```

web/views/user_view.ex - Github

We saw that depending on the version, the controller will either render the `show.v1.json` or `show.v2.json` view. The view also uses pattern-matching to determine the content that will be rendered.

On line `4` and `11` we pattern-match the name of the view that was passed by the controller. When rendering `show.v1.json` view, the JSON structure contains the fields:

- id
- name
- favorite_sport

When rendering `show.v2.json` view, the JSON structure contains the fields:

- id
- name
- favorite_sports

As you can see, both controller and view rely on pattern-matching to render the different versions of the API. It means that to have this part of the application working as expected, we need to set the `:version` value inside the connection assigns, to either `:v1` or `:v2`. It will enable our controller and view to render the right version.

In the next sections you will see how different strategies of API versioning achieve that.

Versioning with request parameter

The first versioning strategy that we’ll cover is the parameter one. Based on the product changes we previously seen, we can imagine the following requests/responses:

Requests to version v1

```
$ curl http://localhost:4000/users/1?version=v1


{
  "name":"John Doe",
  "id":1,
  "favorite_sport":"Ski"
}
```

Requests to version v2

```
$ curl http://localhost:4000/users/1?version=v2


{
  "name":"John Doe",
  "id":1,
  "favorite_sports": [
    "Ski",
    "Sailing"
  ]
}
```

In this example, the value of the `version` parameter is used to determine which version the client is able to handle, therefore rendering a different response to each version.

```
 1 defmodule Profiles.Router do
 2   use Profiles.Web, :router
 3
 4   pipeline :api do
 5     plug :accepts, ["json"]
 6     plug Profiles.Version, %{"v1" => :v1, "v2" => :v2}
 7   end
 8
 9   scope "/", Profiles do
10     pipe_through :api
11     resources "/users", UserController, only: [:show]
12   end
13 end
```

web/router.ex - Github

The web/router.ex file is quite simple and you can see that we have only one scope on line9. This scope uses the api pipeline, that's present on line 4. This pipeline does two things:

- Says we are capable of handling JSON, on line 5
- Executes the plug Profiles.Version on line 6, passing a map to it

Only after executing the pipeline, the router is able to handle the user resource, on line 11.

We know that the controller and the view are expecting the :version to be present as part of the connection assigns value. The code responsible for setting the :version value is can be found in the plug Profiles.Version:

```
1 defmodule Profiles.Version do
2   import Plug.Conn
3
4   def init(versions), do: versions
5
6   def call(%{params: %{"version" => version}}=conn, versions) do
7     assign(conn, :version, Map.fetch!(versions, version))
8   end
9 end
```

web/version.ex - Github

When a request is performed, the function call/2 (on line 6) is called, setting the value of :version in the connection assigns. On line 6 you can see that we pattern-match the request parameters, extracting the version parameter that was sent by the client (query-string). The versions map (passed by in `web/router.ex` on line `6`) is used to determine the `:version` that will be set.

Versioning an API using parameters offers the flexibility to easily switch versions and test the API using the browser. One thing that you need to take care of, is to never use the `version` parameter (used in this example) in a different context, because it could override the API versioning feature.

In the next section, you will see a different strategy, that also brings the flexibility to the API, but that offers a “cleaner” approach.

Versioning with URL

This strategy is really common, being used by the most popular public APIs. It offers the same flexibility we found in the parameter strategy, but offering a more robust approach.

In order to deliver the product changes we had previously discussed, you can imagine this API to look like the following:

Requests to version v1

```
$ curl http://localhost:4000/v1/users/1


{
  "name":"John Doe",
  "id":1,
  "favorite_sport":"Ski"
}
```

Requests to version v2

```
$ curl http://localhost:4000/v2/users/1


{
  "name":"John Doe",
  "id":1,
  "favorite_sports": [
    "Ski",
    "Sailing"
  ]
}
```

With this strategy, the client uses the URL to communicate the version it is able to handle. The`/v1/...` says the client is capable of handling the version `:v1`. Requests to `/v2/...` says the client is capable of handling the version `:v2`. Let's take a look in what the `web/router.ex` looks like:

```
 1 defmodule Profiles.Router do
 2   use Profiles.Web, :router
 3 
 4   pipeline :v1 do
 5     plug :accepts, ["json"]
 6     plug Profiles.Version, version: :v1
 7   end
 8 
 9   pipeline :v2 do
10     plug :accepts, ["json"]
11     plug Profiles.Version, version: :v2
12   end
13 
14   scope "/v1", Profiles do
15     pipe_through :v1
16     resources "/users", UserController, only: [:show]
17   end
18 
19   scope "/v2", Profiles do
20     pipe_through :v2
21     resources "/users", UserController, only: [:show]
22   end
23 end
```

web/router.ex - Github

This time we have two scopes:

- `/v1`, pipes through the `:v1` pipeline
- `/v2`, pipes through the `:v2` pipeline

Both scopes handle the user resource and similarly to the parameter strategy, the pipeline runs the plug `Profiles.Version`, that's the one responsible for setting the `:version` value in the connection `assigns`. You can see on lines 6 and 11 that the `:version` value passed to the plug is hard-coded. It will be used inside the `Profiles.Version` plug:

```
 1 defmodule Profiles.Version do
 2   import Plug.Conn
 3   @versions Application.get_env(:plug, :mimes)
 4 
 5   def init(opts), do: opts
 6 
 7   def call(conn, opts) do
 8     assign(conn, :version, opts[:version])
 9   end
10 end
```

web/version.ex - Github

On line 7, we set the `:version` value to the connection assigns. The value comes from the option `:version` that was set in the `web/router.ex` file.

Similarly to the parameter strategy, versioning an API using the URL offers the flexibility to easily switch versions and test the API in the browser. It also makes the version clear to the API consumer, since all requests will include the version as part of it.

In the next section, we will take a look at the last strategy that is covered by this book.

Versioning with `Accept` header

I personally prefer to use the `Accept` header when versioning APIs. Without changing the resource URI, you can communicate which version you are interested on and also the format. For example, the `application/vnd.app.v1+json` communicates to our API that we want the `:v1` version with a JSON response. We could add support to XML responses, defining a header like `application/vnd.app.v1+xml`.

Considering the product changes that we had previously seen, we can imagine our API requests/responses look like:

Requests to version v1

```
$ curl -H "Accept: application/vnd.app.v1+json" http://localhost:4000/users/1


{
  "name":"John Doe",
  "id":1,
  "favorite_sport":"Ski"
}
```

Requests to version v2

```
$ curl -H "Accept: application/vnd.app.v2+json" http://localhost:4000/users/1


{
  "name":"John Doe",
  "id":1,
  "favorite_sports": [
    "Ski",
    "Sailing"
  ]
}
```

The `Accept` header communicates the version the client is able to support, including also information about the format (in this case JSON). In order to get your Phoenix application working with these custom headers, you will need to add the following lines to your `config/config.exs` file:

```
config :plug, :mimes, %{
  "application/vnd.app.v1+json" => [:v1],
  "application/vnd.app.v2+json" => [:v2]
}
```

We are registering two `mimes`: `application/vnd.app.v1+json` and `application/vnd.app.v1+json`. The first maps to `:v1` and the second to `:v2`. We are going to use them in our `web/router.ex`:

```
 1 defmodule Profiles.Router do
 2   use Profiles.Web, :router
 3 
 4   pipeline :api do
 5     plug :accepts, [:v1, :v2]
 6     plug Profiles.Version
 7   end
 8 
 9   scope "/", Profiles do
10     pipe_through :api
11     resources "/users", UserController, only: [:show]
12   end
13 end
```

web/router.ex - Github

On line 5 we make use of our new registered `mimes`, letting Phoenix know that we only accept `:v1` and `:v2`.

To get it working properly, after adding the lines above to your `config.exs` file, run the following command:

```
$ mix deps.clean plug --build && mix deps.get
```

It will prevent you from finding errors like this:

```
[info] Running ActivityTracker.Endpoint with Cowboy using http://localhost:4000
Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> [info] GET /users/1
[info] Sent 406 in 39ms
...
```

The same way we used the `Profiles.Version` custom plug in the previous two strategies, we are also going to use it here:

```elixir
 1 defmodule Profiles.Version do
 2   import Plug.Conn
 3   @versions Application.get_env(:plug, :mimes)
 4 
 5   def init(opts), do: opts
 6 
 7   def call(conn, _opts) do
 8     [accept] = get_req_header(conn, "accept")
 9     {:ok, [version]} = Map.fetch(@versions, accept)
10     assign(conn, :version, version)
11   end
12 end
```

web/version.ex - Github

On line `3` we set `@versions` with the map we defined in the `config/config.exs` file, where we map different `mimes` to a specific version (either `:v1` or `:v2`). When a request is performed, we extract the value of the `Accept` header on line `8` and then we use the `@versions` map to get the version the value maps to.

Differently from the previous two strategies, this one doesn’t offer you the flexibility to test the API using the browser (unless you use an advanced extension). Besides that, I prefer this strategy when working with APIs, because in my opinion it looks cleaner.

Conclusion

In this book we covered three different approaches on how to version an API using the Phoenix framework. There's no right or wrong way, each strategy offers its own trade-offs. In the end, deciding which strategy is the best option is a technical decision, based on the project needs.

Throughout this book, we changed the strategy three times, using completely different ways to achieve API versioning. Thanks to Phoenix framework design, this task was simple and we limited the scope of the changes to the routing layer.

I would love to hear your feedback about this book and how it helped you. Feel free to send an email to elvio@elviovicosa.com.

Thank you for reading this book and if you want to read more about Elixir and Phoenix, visit my http://www.elviovicosa.com/.