Extracted from:

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

The Pragmatic Bookshelf

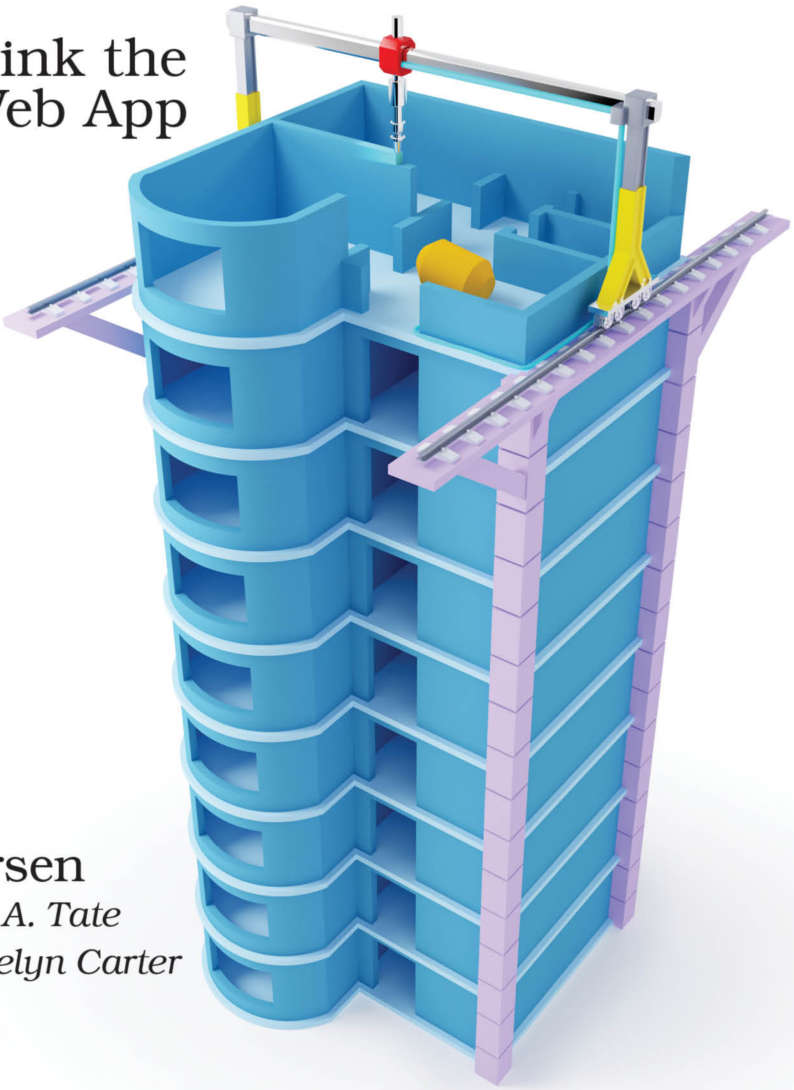
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App



Lance Halvorsen

Series editor: *Bruce A. Tate*

Development editor: *Jacquelyn Carter*

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

Lance Halvorsen

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-243-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

Getting Started with GenServer

GenServers are everywhere in Elixir code. Becoming proficient with GenServer is one of the best things you can do to level up as an Elixir developer. It will require some work on your part. You'll need to learn how client functions, module functions, and callbacks work and interact.

But honestly, implementing a GenServer is pretty straightforward. We'll get lots of practice in this chapter, so you'll come out of it knowing your way around.

Let's begin with a new file in the lib directory called lib/islands_engine/game.ex. This will define a new module that will become our GenServer.

```
defmodule IslandsEngine.Game do
  use GenServer
end
```

By adding the use GenServer line, we already have the beginnings of a functioning GenServer.

The GenServer module defines the start_link/3 and start/3 functions for spawning new processes. They take the name of the module to spawn, an initial state, and an optional list of options.

Let's try it out in the console, specifying our new Game module to spawn as well as an empty map for the state.

start_link/3 will return a tagged tuple—{:ok, <PID>} on success and {:error, <reason>} on failure. We can pattern match on the return and bind a variable to the PID on success.

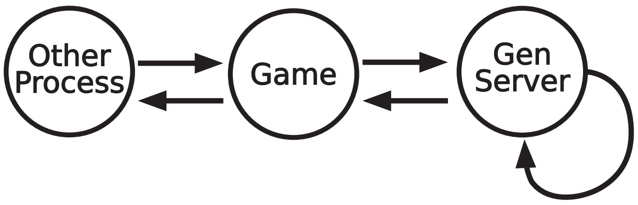
```
$ iex -S mix
. . .
iex> alias IslandsEngine.Game
IslandsEngine.Game

iex> {:ok, pid} = GenServer.start_link(Game, %{})
{:ok, #PID<0.104.0>}
```

Great! We're already able to start the server, and we've hardly written any code.

The GenServer Pattern

There's a simple pattern at the heart of every bit of functionality we build in a GenServer. It has three moving parts—a client function, a function from the GenServer module, and a callback. The client function is the public interface, the part that other processes will call. Within the client function we'll call a

GenServer module function that does some internal work before it triggers a callback. The callback is where we do the real work and return a response.



That's the pattern: a client function wraps a GenServer module function, which triggers a callback. We'll see it again and again, both in GenServers and more generally in other OTP Behaviours.

Client functions hold no surprises. They're just everyday Elixir functions. We can name them whatever we want, and they can take any number of arguments.

GenServer defines its own module functions, so we need to abide by their names and arities. GenServer is specific about callback names and arities as well. We can't invent our own.

There's a direct mapping between GenServer module functions and callbacks. Calling GenServer.start_link/3 will always trigger GenServer.init/1. GenServer.call/3 calls GenServer.handle_call/3, and GenServer.cast/2 maps to GenServer.handle_cast/2.

These three pairs of module functions and callbacks are the ones we'll need to build the GenServer for our game.

:gen_server Callbacks

The Erlang online documentation has a full list of :gen_server module functions and callbacks.[1] In a slightly confusing twist, the docs prepend the callback names with "Module:". These module functions and callbacks handle everything from initializing a process to cleaning up when a process terminates.

Don't worry if this seems abstract at the moment. We'll work through a number of concrete examples in the next few sections.

Passing Messages

The simplest thing we can do with a GenServer is spawn a new server process and send it a message. We've just seen how to spawn a new game server process and bind the resulting PID to a variable. Once we have that PID, we

1. http://erlang.org/doc/man/gen_server.html#Module:code_change-3

can use Kernel.send/2 to send it a message. Once we have message passing down, we can customize behavior based on that message.

Let's see how this all works.

In a new IEx session, let's start a new game process and send it the message :first:

```
iex> alias IslandsEngine.Game
IslandsEngine.Game

iex> {:ok, game} = GenServer.start_link(Game, %{}, [])
{:ok, #PID<0.128.0>}

iex> send(game, :first)
:first
20:49:47.773 [warn]  IslandsEngine.Game #PID<0.128.0>
                     received unexpected in handle_info/2: :first
```

That worked, after a fashion. At least it didn't crash the IEx process.

The use GenServer line we added to IslandsEngine.Game triggers a macro that compiles default implementations for all of the GenServer callbacks into our Game module. That's why we can actually start the ultra-minimal GenServer we currently have. We'll implement new clauses of these callbacks that override the defaults to fit our needs as we customize the game server.

The warning we got is the compiler's way of telling us we need to implement a clause of the handle_info/2 callback to override the default and match the message we sent.

Let's go ahead and define a handle_info/2 clause in our game server that matches the message :first:

```
def handle_info(:first, state) do
  IO.puts "This message has been handled by handle_info/2, matching on :first."
  {:noreply, state}
end
```

The GenServer module itself provides the second argument, state, when it triggers the handle_info/2 callback. state represents the data structure that the individual GenServer process holds. In this case, we defined it as an empty map when we spawned the process.

The return tuple {:noreply, state} tells the GenServer Behaviour that we don't need to send a message back to the caller, and that the value bound to the state variable should become the new state of the GenServer process. In this case, we haven't transformed the state, so it will still be an empty map.

Now we can recompile Game and try again:

```
iex> {:ok, game} = GenServer.start_link(Game, %{}, [])
{:ok, #PID<0.128.0>}

iex> send(game, :first)
This message has been handled by handle_info/2, matching on :first.
:first
```

That's definitely an improvement over our first try.

Now that we have the idea of sending messages to a GenServer process, let's add a little complexity.

Introducing Calls

More often than not, we're going to want a meaningful response when we send a GenServer process a message. We might query the process's state, or we might want to see the result of a command we've sent it. This is where calls come in.

GenServer calls are synchronous. They can return any arbitrary value to the caller, and if the caller is waiting for a return, it will block until it gets one. The GenServer callback that handles calls is handle_call/3. It's similar to handle_info/2 in that it pattern matches for a message as its first argument.

It's different from handle_info/2 in that it doesn't accept messages sent directly from other processes. Instead, it's triggered whenever we call GenServer.call/2.

Let's try this out. In lib/islands_engine/game.ex add a clause of handle_call/3 that looks like this one. Our aim is to simply have it return the initial server state.

```
def handle_call(:demo_call, _from, state) do
  {:reply, state, state}
end
```

The key here is the first argument, :demo_call. This is the pattern that will determine which clause of handle_call/3 to execute. We'll see where it comes from shortly.

Don't worry about the other arguments. GenServer itself will provide them internally.

The return value is different from the one we used in handle_info/2. It indicates that we'll be replying to the caller. The middle element is the actual reply, and the third element is what we want the state of the GenServer process to be.

Now let's go back to the IEx session we had going and recompile the Game module. Then let's start a new server with %{test: "test value"} as the initial state. Make sure to pattern match on the return so we'll bind the PID to the game variable.

```
iex> {:ok, game} = GenServer.start_link(Game, %{test: "test value"})
{:ok, #PID<0.130.0>}
```

Now invoke GenServer.call/3 with the PID and the atom :demo_call that we specified as the first argument to our clause of handle_call/3. We should get back the state we set when we started the process.

```
iex> GenServer.call(game, :demo_call)
%{test: "test value"}
```

Success! We got the initial state back.

When we invoke GenServer.call/3, GenServer keeps track of the second argument we passed, grabs the PID of the calling process, and gets the process's state. Then it invokes GenServer.handle_call/3 with those arguments, in order:

```
def handle_call(:demo_call, _from, state) do
```

_from is a tuple that contains the PID of the calling process, the IEx session in our case. We could use it to send messages back to the caller, but we don't need to here, so we prepend it with an underscore.

Wait a Minute…

Our callback returned a tagged tuple, but we only saw the server state in the console. That's because the GenServer processed our callback's return value internally in order to formulate a final reply to the caller. It stripped out the :reply tag and used the final state element to set the new state in the GenServer.

In order to expose this functionality as part of the public interface, we need to define a client function to wrap GenServer.call/3 in lib/islands_engine/game.ex. The only argument it needs is the server PID.

```
def demo_call(game) do
  GenServer.call(game, :demo_call)
end
```

This should behave exactly the same as using GenServer.call/3 directly. Let's try it out. We'll need to recompile the Game module or else start a new session and alias IslandsEngine.Game.

```
iex> {:ok, game} = GenServer.start_link(Game, %{test: "test value"})
{:ok, #PID<0.125.0>}

iex> Game.demo_call(game)
%{test: "test value"}
```

It returns the server state, which is just what we want.

Introducing Casts

Casts work a lot like calls, so this section will seem familiar. The difference is that casts are asynchronous; they don't return a specific reply, so the caller won't wait for one.

Casts can increase throughput if synchronous processing becomes a bottleneck. But we should prefer calls to casts because they provide a kind of back pressure, limiting the amount of work a process will accept at any given time and preventing it from getting overloaded.

It's good to know how to use casts, though, so we'll practice writing one here. Let's start by defining a handle_cast/2 callback.

We'll have it take a tuple containing the atom :demo_cast as well as a new value we want to set in the state. Then we'll use the Map.put/3 to set a new value for the state's :test key.

Casts don't reply to the calling process, so GenServer won't pass in a reference to it into handle_cast/2.

```
def handle_cast({:demo_cast, new_value}, state) do
  {:noreply, Map.put(state, :test, new_value)}
end
```

We'll return a tagged tuple as our handle_call/3 did. We won't need to reply to the caller, so it will only have two elements—:noreply and the new server state.

To set this up, let's start up a new GenServer and call Game.call_demo/1 with the PID to check the state we have:

```
iex> {:ok, game} = GenServer.start_link(Game, %{test: "test value"})
{:ok, #PID<0.130.0>}

iex> Game.demo_call(game)
%{test: "test value"}
```

We get the initial state back, which is what we expect.

Now let's run the cast, followed by the call to return the state. If all goes well, we should get the new state back:

```
iex> GenServer.cast(game, {:demo_cast, "another value"})
:ok

iex> Game.demo_call(game)
%{test: "another value"}
```

Indeed, the cast did work.

We can wrap the GenServer.cast/2 call in a client function, and it should behave the same as the bare GenServer.cast/2 call.

```
def demo_cast(pid, new_value) do
  GenServer.cast(pid, {:demo_cast, new_value})
end
```

Now that we have the basics down, we can delete the handle_info/2, handle_call/3, and handle_cast/2 callbacks as well as the demo_call/1 and demo_cast/2 functions. We won't need them for the rest of our work here.