Extracted from:

Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

This PDF file contains pages extracted from *Functional Web Development with Elixir, OTP, and Phoenix*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

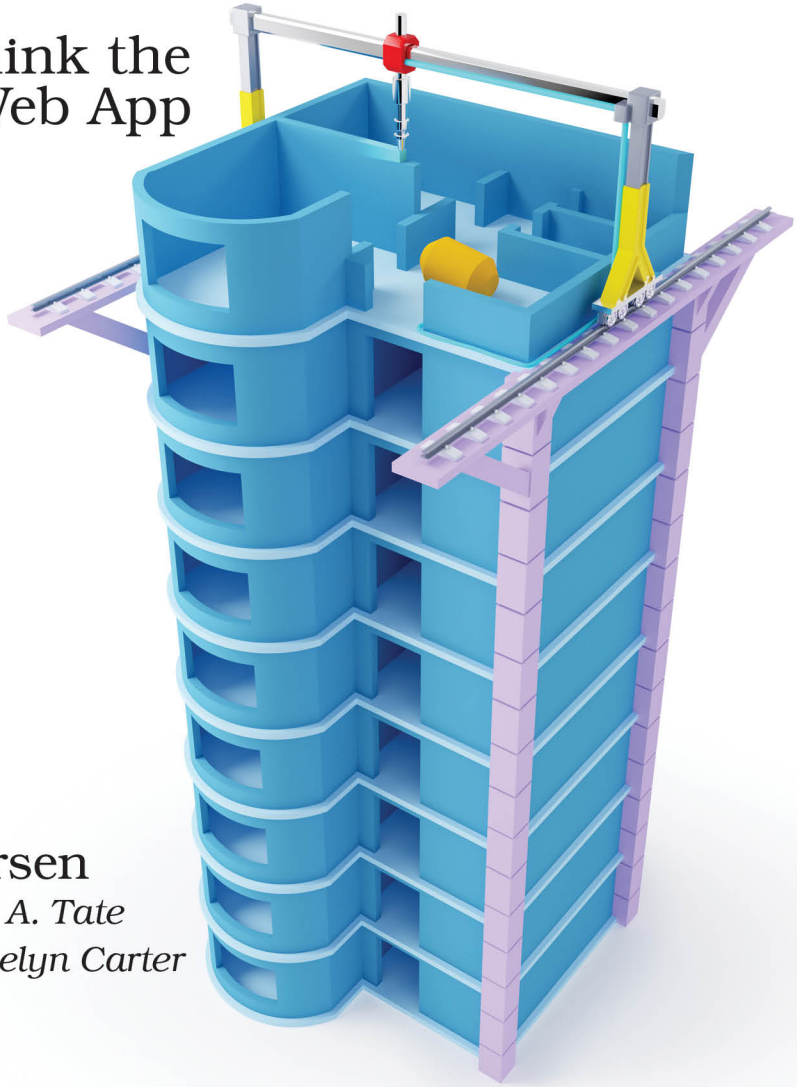
Raleigh, North Carolina

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*

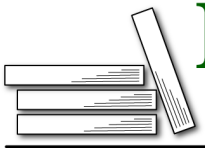
Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

Lance Halvorsen

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

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

Copyright © 2017 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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: B1.0—March 22, 2017

Getting Started With GenServer

GenServer is an Elixir module that wraps its Erlang counterpart, :gen_server. GenServer automatically creates default implementations of the :gen_server callbacks so that we only write code specific to our GenServer. We'll use the Elixir GenServer wrapper as we build our game server, which will spare us from writing a lot of boilerplate.

In the last chapter we saw that Agents are processes designed to handle state. Elixir Tasks are processes designed to take action, to do things in the system. Both of these are built on top of GenServer. They are specialized forms of GenServers, and GenServers are capable of both managing state and taking action.

Since GenServer is an OTP Behaviour, Agents and Tasks are OTP Behaviours deep in their core. Elixir abstracts away most of their OTP heritage, so we hardly need to know anything about OTP to use them.

GenServer will require more of us. We'll need to learn how client functions, module functions, and callbacks work as well as how they work together.

But honestly, GenServer is pretty straight-forward. We'll get lots of practice working with it in this chapter, so you'll come out of it knowing your way around.

Our task in the next few sections is to develop a sense of how GenServers work. We'll do that by building out a customized GenServer for Islands. All the code for interacting with the game will live in this module. It's a great example of the focus we get by using GenServers.

Let's begin with a new file in the lib directory called lib/game.ex. This will define a new module which 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, just as Agents do. They take the name of the module to spawn, an initial state, and an optional list of options.

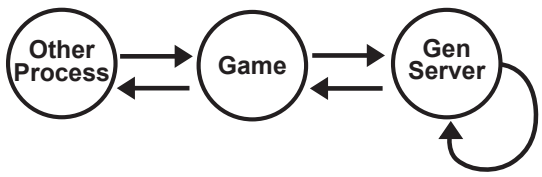
Let's try it out in the console.

```
~/work/islands_engine$ iex -S mix
```

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

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

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 talked a bit about message passing before, but this is our first chance to really show it in action.

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

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

iex> send(game, :first)
:first
```

Well, that worked, but it didn't do much. It just returned the message to us. Let's see if we can make this a little more interesting.

GenServer allows us to define a callback that will handle arbitrary messages and specify behavior for any given message the process receives. The callback is handle_info/2, and the first argument is the message. We can define multiple

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

clauses of handle_info/2 if we need to, with different messages, and the normal rules of pattern matching will determine which clause gets executed.

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.

Now we can recompile IslandsEngine.Game and try again.

```
iex> {:ok, game} = GenServer.start_link(IslandsEngine.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. Of course, we can define any arbitrary behavior we'd like in that clause.

What happens if we send another, different message?

```
iex> send(game, :second)
:second
```

We get the same behavior as we did before we implemented our clause of handle_info/2.

This works the way it does because GenServer provides a default clause for handle_info/2 that matches any message, something like this.

```
def handle_info(_, state) do
  {:noreply, state}
end
```

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 which override the defaults to fit our needs as we customize the game server.

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' 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/3 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/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, _from, state) do
  {:reply, state, state}
end
```

Then, 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(IslandsEngine.Game, %{test: "test value"})
{:ok, #PID<0.130.0>}
```

Now invoke GenServer.call/3 with the PID and the atom :demo.

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

Success! We got the initial state back.

When we invoke GenServer.call/3, the GenServer does some work behind the scenes. It keeps track of the second argument we passed it for pattern matching later. It grabs the PID of the calling process, and it gets the current state from the specific GenServer referenced by game. Then it invokes GenServer.handle_call/3 with those arguments, in order.

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

_from is 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/game.ex. The only argument it needs is the server PID.

```
def call_demo(pid) do
  GenServer.call(pid, :demo)
end
```

This should behave exactly the same as using GenServer.call/3 directly. Let's try it out.

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

iex> IslandsEngine.Game.call_demo(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.

They can come in handy if sequential processing turns into a bottleneck, but in general, Elixir developers prefer calls to casts, even if the calls just return :ok. 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. handle_cast/2 clauses take two arguments instead of three. They don't reply to the calling process, so they don't need its PID. Let's have ours take the atom :demo as well as the server state.

```
def handle_cast(:demo, state) do
  {:noreply, %{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. In this case, we'll change the state a little bit so we can see it work.

To set this up, let's start up a new GenServer and do a GenServer.call(pid, :demo) to check the state we have.

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

iex> GenServer.call(game, :demo)
%{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(pid, :demo)
:ok

iex> GenServer.call(pid, :demo)
%{test: "new 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 cast_demo(pid) do
  GenServer.cast(pid, :demo)
end
```

Now that we have the basics down, we can delete the handle_info/2 and handle_cast/2 callbacks as well as the cast_demo/1 function. We won't need them for the rest of our work here. The handle_call/3 callback and call_demo/1 function will come in handy for the rest of this chapter, but we should delete them if we ever deploy this application. Having a pair of functions that dump out all the server's state is not a great practice for a production application.