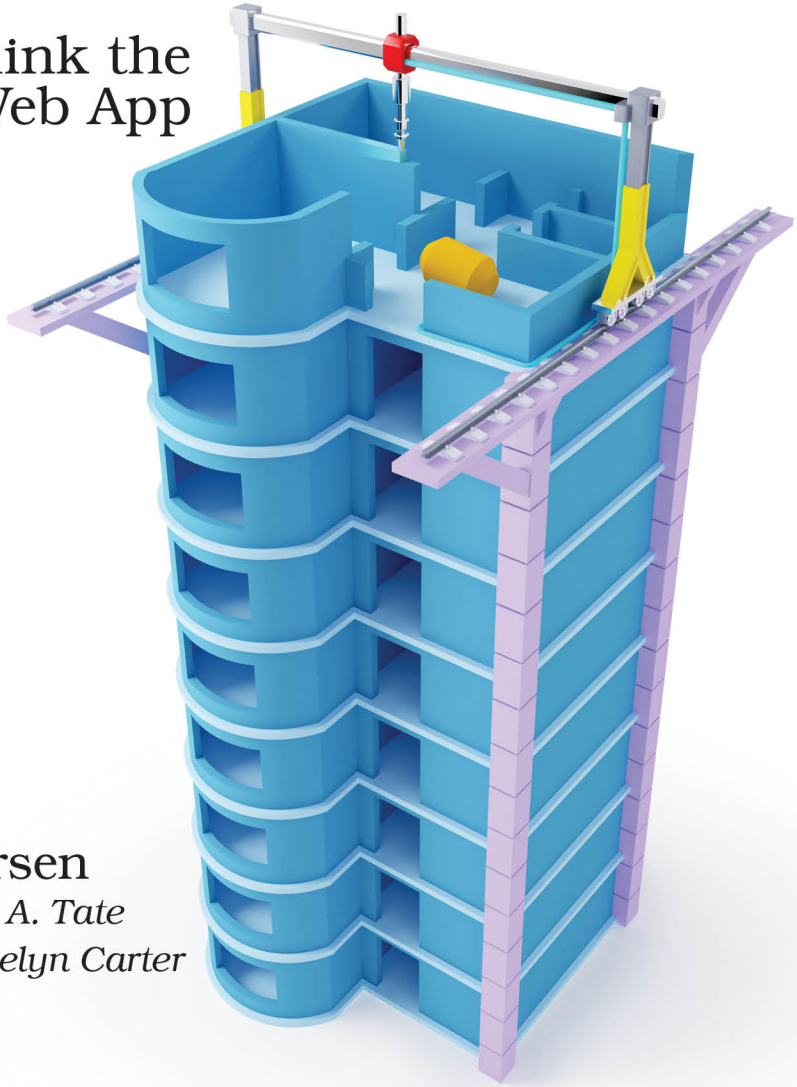
The Pragmatic Programmers

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*

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/lhelph/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Andy

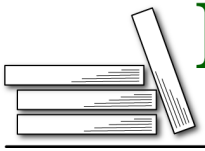
Functional Web Development with Elixir, OTP, and Phoenix

Rethink the Modern Web App

Lance Halvorsen

The Pragmatic Bookshelf

Raleigh, North Carolina



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*.

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: B2.0—April 11, 2017

Contents

Part II — Add a Web Interface With Phoenix

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent changes first.

Beta 2—April 11, 2017

- Added Chapter 4, *Manage State with gen_statem*, on page 83
- Addressed errata

Beta 1—March 22, 2017

- Initial release

Preface

Elixir, OTP, and Phoenix make a beautiful and powerful team. Throughout this book we'll be building a web application together that shows what they can really do when used well together.

We'll see their beauty as we create cleanly separated application layers with clear responsibilities.

We'll see their power as we tackle long standing problems in web development, and move out into new directions.

Most of all, we'll have fun, and learn things we can use to make the code for your day job or side projects more beautiful, easier to maintain, and simply a joy to work on.

Who This Book is For

On a practical level, this book is for people who have some familiarity with Elixir and Phoenix, and who want to take that knowledge further. But there's a wider list for whom the ideas in this book will resonate.

For people who view OTP with a little trepidation, or for those who haven't quite mastered OTP Behaviours, this book will give you the confidence to use OTP in any application.

For people who have felt the sting of tight coupling between business logic and web frameworks, this book will show you a way out of that pain forever.

For people who feel constrained by traditional web development, you'll learn new techniques and new ways to structure apps that will spark your imagination.

For people who are wondering what all the fuss is about with Elixir and Phoenix, you'll get a great taste of what makes people so excited. You just might become a convert!

How to Read This Book

The book is ordered around building an application in sequential layers. If you're planning on implementing the game as you read, which is a great idea, it's important to work through the chapters in order.

If you're the sort of reader that likes to skip around, though, all is not lost. You can read the first few sections of any chapter—up until where we start to really implement the code—in any order, and they will still hold value.

Online Resources

The code we'll develop is available at the Pragmatic Programmers site for this book. There's also a community forum and errata-submission form for you to ask questions, report any problems with the text, or make suggestions for future versions.[1]

In addition, there will be an organization on github devoted to community-supplied frontend implementations, in a variety of different languages and frameworks, for the application we're about to build.[2]

1. https://pragprog.com/book/lhelph/functional-web-development-with-elixir-otp-and-phoenix
2. https://github.com/functional-web-dev

CHAPTER 1

Mapping Our Route

Welcome! We're about to go exploring, and it's going to be a blast. We're going to do what many of us say we love most—play with new languages, experiment with new techniques, and expand our understanding of writing software for the Web. Whenever you go exploring, it's important to have a map, a good idea of where you're headed, and a plan for how you'll get there. That's what this chapter is all about.

Many early client server systems were stateful. Servers kept working state in memory. They passed messages back and forth with their clients over persistent connections. Think of a banking system with a central mainframe and a dedicated terminal for each teller. This worked because the number of clients was small. Having fewer clients limited the system resources necessary to maintain those concurrent connections.

Then Tim Berners-Lee invented a new client server system called the World Wide Web.

The Web is an incredibly successful software platform. It's available almost everywhere on Earth, on virtually any device. As the Web has grown and spread, so has HTTP. HTTP is a stateless protocol, so we think of Web applications as stateless as well. This is an illusion. State is necessary for applications to do anything interesting, but instead of keeping it in memory on the server, we push it off into a database where it awaits the next request.

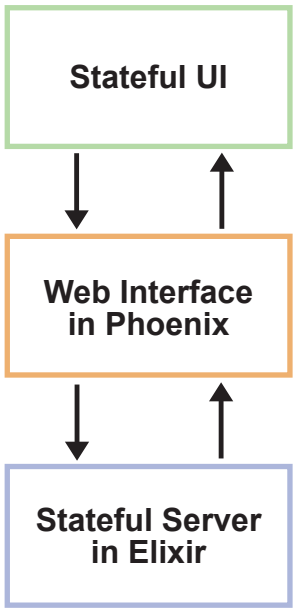
Offloading state to a database provides some real advantages. HTTP based applications need only maintain temporary connections with clients until they send a response, so they require far fewer resources to serve the same number of requests. Most languages can't muster the concurrency necessary to maintain enough persistent connections to be meaningful for a modern Web application.

Going “stateless” has let us scale.

But statelessness comes at a cost. It introduces significant latency as applications need to make one or more trips to the database for the data to prepare a response. It makes the database a scaling bottleneck, and it habituates us to model data for databases rather than for application code.

Elixir offers more than enough concurrency to power stateful servers. Phoenix Channels provide the conduit. A single Phoenix application can maintain persistent Channel connections to hundreds of thousands or even millions of clients simultaneously. Those clients can all broadcast messages to each other, coordinated through the server. While processing those messages, the application remains snappy and responsive. Elixir and Phoenix provide a legitimate alternative to stateless servers capable of handling modern Web traffic.

We’re about to explore this new opportunity with a stateful application written in Elixir and a persistent Phoenix Channel ready to connect it to any frontend application.



We’ll do this by building a game called Islands. It may not be a top download on your favorite gaming platform, but it will be fun to play. Most importantly, you’ll learn a lot by building it. Game developers have always pushed the web to the extreme. They’ve had to approach problems in novel ways to meet their

performance needs. We'll be re-thinking our approach as well, and what we'll learn will help us solve everyday business problems in radically improved ways.

We're going to tackle Islands in distinct parts. We'll start with a stateful game engine written in Elixir, and then we'll layer on a web interface with Phoenix. We'll stop just short of building out a full frontend application—there won't be any new territory for us to cover—but we'll provide examples for you to run and follow along with.

We'll build Islands in a way you might not be used to, so let's get an idea of what lies ahead.

Lay The Foundation With Elixir

In Part One we'll begin by building a stateful game server in pure Elixir. We won't use a database to store the game state, and we'll define our domain elements with native Elixir data structures instead of ORM models. We'll maintain state with Elixir Agents, and we'll wrap those agents up in a GenServer to provide a consistent interface to the game.

We'll bring in a finite state machine to manage state transitions—like switching from one player's turn to the other, and moving from a game in progress to one player winning. We'll also build a supervision tree to take advantage of Elixir's incredible fault tolerance.

Building the game engine solely in Elixir solves a long standing problem in Web development, the tendency for framework code to completely entangle application logic so the two can't be easily separated. Without that separation, it's hard to reuse application logic in other contexts. As we build Islands, we won't even begin to work with the Phoenix framework until our game logic is complete.

By the time we're done with Part One, we'll have a fast, fault tolerant game engine that can spin up a new GenServer for a game almost instantly. We'll be able to reuse it with any interface we want—the Web, a native mobile app, plain text, or whatever else we can think of. If we look at it the right way, the GenServer for each game is really a microservice, or a nanoservice, living right inside the virtual machine.[1]

1. http://blog.plataformatec.com.br/2015/06/elixir-in-times-of-microservices/

Add a Web Interface With Phoenix

In Part Two, we'll generate a new Phoenix application without Ecto, the database layer that ships with Phoenix. We'll bring in our new Islands engine as a dependency and make it part of our new Phoenix application's supervision tree. We'll create a lobby for the game with the standard Phoenix MVC parts—the router, a controller, a view, and some templates.

Then we'll move on to the really exciting part, replacing HTTP's temporary client-server connections with persistent ones via Phoenix channels. Channels provide a conduit for lightning-fast message passing between front end applications, and in our case, a stateful back end server. We'll make good use of Channel naming conventions to allow two players to connect to their own private GenServer running Islands. And we'll be able to run thousands of games simultaneously on a single server. Many languages would struggle to keep persistent connections open for all the players of all current games, but Elixir's incredible concurrency model will make it easy.

As we finish up, we'll have a Web interface to our Islands engine. The main component will be a Phoenix Channel able to connect two players directly to an individual Islands game. We'll customize the JavaScript files that Phoenix provides to get it primed and ready for your favorite frontend framework. When we're done, it'll have much less code and far fewer moving parts than a conventional Web application.

Functional Web Development

With all this in mind, you may be wondering about the title of the book and how this represents *functional* web development.

One of the most characteristic patterns of functional programming is composition. With function composition, we take a big, complex piece of work and split it up into smaller, decoupled, and more focused functions. Then we recreate the full behavior by chaining these functions together. This not only helps us reason about our programs because smaller functions reduce cognitive load, but it helps with maintainability because smaller functions are easier to work on.

We'll see this again and again when we work with Agents and GenServer.

In this book, we'll take the idea of composition from the level of functions and scale it up to the level of applications. We'll take the full, complex behavior of a web application and separate it into independent, decoupled layers. Each layer will have a focused responsibility. It will do its job and nothing else.

Then we'll recreate the full behavior of the application by having each layer call into the next, passing the return values back up the chain and out to the client. By doing this, we'll gain clarity and maintainability for our whole application.

Now we're ready to introduce the game itself.

The Game of Islands

Let's talk a little bit about Islands. It's a game for two players, and each player has a board which consists of a grid of one hundred coordinates. The grid is labeled with the letters A through J going down the left side of the board and the numbers 1 through 10 across the top. We name individual coordinates with a letter-number combination—A1, J5, D10, and so on.

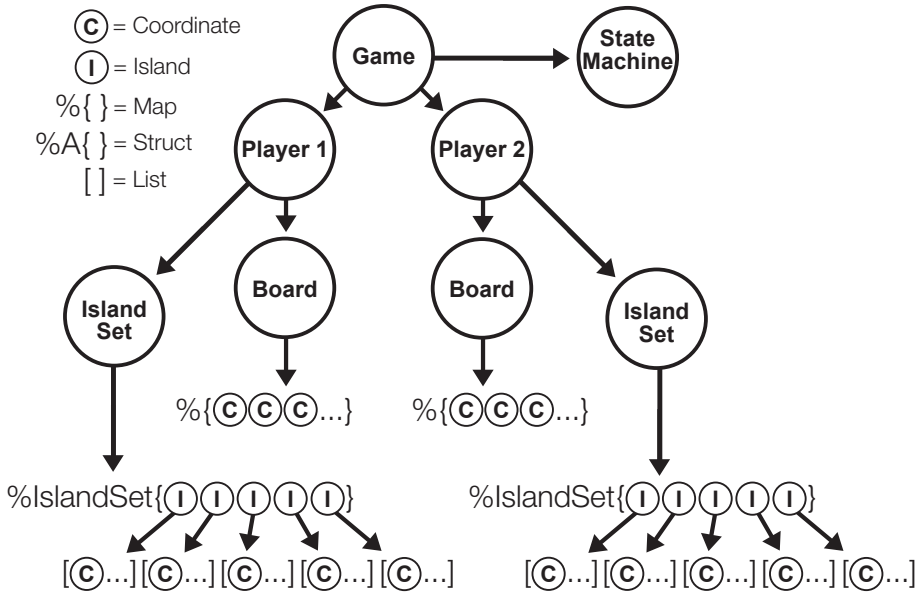
The players cannot see each other's boards.

The players have matching sets of islands of various shapes and sizes which they place on their own boards. The players can move the islands around as much as they like until they say that they are set. After that, the islands must stay where they are for the rest of the game.

Once both players have set their islands, they take turns guessing coordinates on their opponent's board, trying to find the islands. For every correct guess, we plant a palm tree on the island at that coordinate. When all the coordinates for an island have palm trees, the island is forested.

The first player to forest all their opponent's islands is the winner.

That gives us a rich, interesting data structure model across multiple processes. When we're done, we'll have something that looks like this.



Before we get to work, let's make sure we have all of our dependencies installed. For the first part of the book, all we'll need are Elixir and Erlang. For the second part, we'll need to install the Phoenix archive, Node.js, and npm. Have a look at the *the (as yet) unwritten appendix.installation_instructions* for help getting them installed.

We've got a plan! Time to start building.

Part I

Build a Game Engine in Pure Elixir

In this first part we will build all the logic for our game in a new Elixir project. We'll package it up as an Elixir application which we can use and re-use in any other project we like. Elixir makes this easy and natural with the use of the Application Behaviour.

What we'll do in this chapter

- *model domain elements as Elixir datatypes*
- *hold state in memory with Agents*
- *write functions to manipulate state*

CHAPTER 2

Model State With Agents

The BEAM, Erlang's virtual machine, is an inherently stateful system. Every application we write in Elixir can be stateful. Every web interface we build with Phoenix can be stateful as well. This is true whether we recognize it or not, and it's true whether we make best use of it or not. We can use the BEAM's stateful nature to our advantage as we model the domains of any of our Elixir applications.

Nearly all Web applications we build today are stateless. They store application state in a database, access it on each request, and write any changes back to the database. We typically define domain entities with ORM models, classes that interact directly with tables in a database.

We're going to adopt a different approach to modeling application state in this chapter. We're going to break our database habit and learn to model domain entities in a more elegant way.

We'll take a look at Elixir processes and how they hold state. We'll see how easy it is to hold and manipulate state with Elixir Agents. Then we'll learn how to model relationships between agents so we can handle complex domains.

Most importantly, we'll change the way we think about state and domain modeling.

Embracing Stateful Servers

Holding state on the server is an old idea. It's the way developers built the earliest client-server applications. We moved to stateless servers with the rise of the Web. That let us scale, but the resulting complexity and latency are pointing us back to our stateful roots.

Stateful servers keep the data they need persistent in memory over the span of many requests. This is very different from the stateless request-response of HTTP.

Think about a typical HTTP GET request that includes an id for a resource. To handle the request, the server will use the id to fetch data for that resource from the database. That data becomes the working state of the application, but only for the fraction of a second it takes to send a response. Immediately after sending the response, the server clears out all that state in preparation for the next request.

In Erlang and Elixir, we can spawn processes that persist in memory for as long as we want. They can hold state and do work. We can write public functions for these processes that allow us to query and manipulate that state, as well as do any other work we'll need.

These processes are ultra-lightweight. We can spawn hundreds of thousands, or even millions of them if we need to. With that in mind, we can begin to shift the responsibility for holding state from the database, to multiple long-lived processes in the virtual machine.

This opens up a whole new world to us in terms of how we model our application domains. We'll be able to work in a way that's natural for the application, not the database. This means we'll use common data structures like lists and maps to replace foreign keys and join tables to model relationships between entities.

The Upside

By dividing up state and storing it in separate, long-lived processes, we'll be separating concerns. Each process will have only the functions it needs to work with the specific type of data it holds. This means that in order to work with data that spans multiple processes, those processes will need to coordinate. We'll pass messages between these processes to query and manipulate state.

Since the application won't need to go across the network to the database each time it needs to fulfill a request, we'll shed latency as well.

We'll design our system so that each piece of state will live in only one place, inside a single process. Each piece of state will have a single representation in the system. This is very different from most Web applications where multiple requests for the same resource can each spawn an object that represents the

same piece of data. Having multiple representations can lead to race conditions and stale data errors.

Using processes in this way can minimize race conditions. Processes communicate by passing messages to one another. Each process has a mailbox that handles these messages in the order they are received. Mailboxes act as a synchronizing mechanism in an otherwise asynchronous system.

Elixir provides an easy to use abstraction for spawning processes to hold state called Agents. We'll use Agents to model the basic building blocks of the game of Islands.

The game of Islands has five main entities: coordinates, boards, islands, players, and the game itself. We've also described a few actions—position islands, fix their positions, and guess a coordinate. In this chapter, we'll build coordinates, islands, boards, and players as well as lay the foundation for the actions.

To do that, we'll need to determine what data each of our domain entities needs. We'll need to find the right data structures to hold that data, and then define public functions that let us query and manipulate it.

Then we'll need to figure out the relationships between the pieces of data. We'll model these relationships with data structures to build bigger structures that more accurately reflect the needs of the game.

For each entity we model, we'll define a separate module, define a function to start a new process with a default data structure as its state, and add public functions to it to define its behavior.

We'll learn to start a new Agent with a default state, get that state back, and update it when we need to. We'll also work out the functions we'll need in order to play the game. When we're done, we'll begin to see Islands take shape.

Now that we know where we're going, let's get a better understanding of processes, state, and Agents.

Elixir Processes

If you've spent any amount of time working with Elixir, you probably heard about processes. Elixir processes are ultra-lightweight, virtual machine-level processes that are completely independent. They share no memory with one another.

Each process does its work asynchronously, independent of any other process' work. The Erlang virtual machine has special OS level threads that act as

schedulers for the virtual machine level processes. The schedulers do their best to use all the cores of your machine as efficiently as possible.

Processes can exist for as long as the virtual machine is running, or they can be ephemeral, terminating and having their memory garbage collected as soon as their task is done. For holding state, long-lived processes are the ticket, and we'll be focusing on them here.

Since there can be many, many separate processes all working asynchronously, they need to coordinate with one another to accomplish complex tasks. They do this by sending messages to each other. The way they specify a given process to send a message to is by a first class data type called a PID, a process identifier.

Each process determines what its own response will be to a given message, even if that response is to ignore the message.

Processes all have mailboxes where they receives messages in a queue, then processes them in order. This is really important because it means that the mailbox is a synchronizing mechanism in an otherwise asynchronous system.

The last property about processes that interests us here is that they can hold state. Let's take a closer look at that now.

Processes and State

Elixir is a functional language, and it handles state in a functional way, with recursion and data transformation.

We can think about state on two levels. On the application level, state means all the data that an application needs to do its work. This could be a single, huge data structure, or more likely, many smaller bits of data. We can talk about those smaller bits of data as state as well. Those are the ones we'll focus on here.

If we separate state into processes correctly, we can ensure that there is a single representation for each piece of data in the whole system. The process that holds the data is the canonical representation of the data. For example, to find one particular player in one given game of Islands in a system with thousands of running games, there should be only one place to look. Different versions of that data should not be spread around the system in different places.

This is utterly unlike the way data models for most web frameworks operate. Multiple requests for the same resource in other systems will generate multiple representations of that data. It's extremely easy for these multiple represen-

tations to get out of synch with each other and cause race conditions and stale data errors.

While synchronization is good, and can make our lives simpler, single processes that handle a lot of requests can also become bottlenecks. Always benchmark!

Now we come to the way that Elixir actually holds and manipulates state.

Every long lived process that holds state in Elixir relies on an infinitely recursive function that takes a data structure as its argument. That data structure *is* the state. Each recursion might or might not transform the data before passing it back to itself for the next recursion.

Let's look at a minimal example to get a sense of this. Every Elixir process that handles state will have a structure something like this one:

```
def loop(state) do
  new_state =
  receive do
    :a_transforming_message -> transform(state)
    anything_else -> state
  end

  loop(new_state)
end
```

The outermost layer is the loop/1 function. This is a plain Elixir function, not a loop like while or for in other languages. It takes a single argument—the state for the process to hold.

Within loop/1 we have a receive block.

The receive block's job is to pick up messages from the processes mailbox in the order they were received, pattern match on them, and execute code which matches the pattern. If there are no messages in the mailbox, the receive block will simply become idle.

Message code executed in the receive block can use the current state of the process to create new state. At the end of the receive block, there is a recursive call to loop/1 with the new state as the argument.

The presence of a new message in the process' mailbox triggers a recursion. Without a new message, the receive block will remain idle and the next recursion of loop/1 will not happen. If there is no change to the current state, we'll simply use it as the "new state" for the recursive call. This new state then becomes the state of the process.

Elixir State is Separate From Behavior

Processes may hold data structures as state, but they never mix state and behavior together. Functions don't automatically have access to state. Either we need to pass it in, or the function needs to go get it. This is a key difference between Elixir and most popular Object Oriented languages.

Now that we've seen the recursive magic trick that maintains Elixir state, let's take a look at Agents, an Elixir abstraction that hides the magic trick and makes our job easier.

Agents

Elixir Agents abstract away the complexities of managing state with Elixir processes. We won't need to write our own recursive functions that query the mailbox for new messages. The whole reason Agents exist is to hold state and allow us to query and manipulate it.

Agents are the simplest way to use *OTP* in Elixir. OTP stands for Open Telecom Platform. It's the Erlang super-library/design pattern combination that makes Elixir and Erlang's incredible concurrency, fault tolerance, and distribution possible.

Elixir abstracts away almost all of the OTP patterns when we use an agent. Under the hood, though, OTP is there. As we move through the next few chapters, we'll gradually see more and more of the inner workings of OTP. For now, agents will give us the gentlest possible introduction.

The Agent module itself provides a few simple functions that allow us to start a new process as well as access and update the agent's state

In order to build an agent, we'll begin with a plain Elixir module. To add new behavior, we'll define public functions that wrap Agent module functions. For each agent we build, we'll have one public function that will start a new agent process with some initial state.

We can write other public functions to query the agent's state and return responses to the caller. Public functions can transform the agent's state as well.

All the public functions will take a PID as their first argument. This provides an address to send the messages to.

An agent's state can be any valid Elixir data type—a string, integer, list, map or struct. Often you'll see the state defined as a struct in the agent module. This will appear as a struct named after the agent module.

We have flexibility surrounding which data structures we can use to hold the agent's state, and we can use this flexibility to our advantage when modeling relationships between agents. An agent's state might be a list that holds other agents. In turn, those agents might hold maps of yet more agents. This is how we can model larger and more complex data structures.

Now that we have a better understanding of processes, state, and agents, we're ready to begin. The first thing we'll need is a new project for our application. That's where we're headed next.

Let's Build It

Enough talk, let's get started!

We'll begin by creating a brand new, supervised Elixir application called `islands_engine` with `mix new islands_engine --sup`. We'll talk more about process supervision in *the (as yet) unwritten chapter.design_for_recovery*. For now, think of it as the mechanism that provides the tremendous fault tolerance we get from the BEAM.

```
mix new islands_engine --sup
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/islands_engine.ex
* creating lib/islands_engine/application.ex
* creating test
* creating test/test_helper.exs
* creating test/islands_engine_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

    cd islands_engine
    mix test

Run `mix help` for more commands.
```

Let's take a look at what mix generated for us.

```
$ tree
.
├── README.md
```

```
├── config
│   └── config.exs
├── lib
│   ├── islands_engine
│   │   └── application.ex
│   └── islands_engine.ex
├── mix.exs
└── test
    ├── islands_engine_test.exs
    └── test_helper.exs

4 directories, 7 files
```

We already have the skeleton of a standard Elixir OTP application.

The `mix new` task created a directory for configuration, a directory for tests, and the `lib/` directory for our application code. The `lib/islands_engine/application.ex` file defines the application. Later, when we build in fault tolerance with a supervision tree, this is where we'll do it.

We're going to model the most central entities for Islands in this chapter, but we're not going to approach them as we normally might in a Web application. We won't talk about foreign keys, many to many relationships, or join tables. Instead, we're going to think in terms of Elixir data structures and composition. Coordinates hold the majority of the state we need to play the game. A list of coordinates makes an island, and a map of coordinates makes a board.

Model A Simple Entity

When modeling a domain, a good approach is to start with the most basic entity. With that, it's possible to compose these basic entities together into more and more complex ones. It's analogous to starting with simple bricks and ending up with walls, columns, and arches.

Coordinates are the core entities with which we'll compose the others in Islands. Let's start our data modeling with them.

Individual Coordinates will hold the state of each individual square on the game board. They really only need to keep track of two things: whether the coordinate is part of an island, and whether a player has guessed it. From those, we can derive the third piece, whether the coordinate has been hit.

A coordinate won't need to keep track of which position it occupies on the board. The board itself will take care of that mapping.

An Elixir Struct is ideal to model a coordinate. Our Coordinate struct will need two keys, `:in_island` to hold an atom representing the island a coordinate

belongs to, and :guessed? to hold a boolean showing whether a player has guessed that coordinate.

When to Use Structs

Structs are a special form of maps where we specify all the keys the struct will have when we define it. This makes it ideal for situations where we need a data structure with a small number of fixed keys, as we have here.

We define structs with the defstruct/1 macro, and we can only do that from inside a module. With that definition set, we can refer to structs of that type by the module name. For example, if we defined a struct within a User module, we would refer to user structs as %User{}.

Start With a Module

For the Coordinate module, we'll need a new file in the lib directory called lib/coordinate.ex. We'll namespace the module with our application name, IslandsEngine, to avoid collisions.

A new Coordinate struct will not be in any island, and neither player will have guessed it. That means the default values should be :none for the :in_island, and false for :guessed?.

```
defmodule IslandsEngine.Coordinate do
  defstruct in_island: :none, guessed?: false

  alias IslandsEngine.Coordinate
end
```

We also alias the module name so that we can define coordinate structs as %Coordinate{} instead of %IslandsEngine.Coordinate{}.

Let's experiment a little with the module and struct that we just created. We can run the application interactively in an IEx session with iex -S mix.

```
$ iex -S mix
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe]
              [kernel-poll:false] [dtrace]

Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)
```

As we did in the module, let's alias IslandsEngine.Coordinate in the console to reduce typing.

```
iex> alias IslandsEngine.Coordinate
IslandsEngine.Coordinate
```

Just typing out an empty coordinate struct evaluates to a new struct with the default keys and values.

```
iex> %Coordinate{}
%IslandsEngine.Coordinate{guessed?: false, in_island: :none}
```

We can use the dot notation to get access to the values for the two keys we defined.

```
iex> %Coordinate{}.in_island
:none
```

```
iex> %Coordinate{}.guessed?
false
```

We can bind the value of that expression to a variable with the match operator, =.

```
iex> coord = %Coordinate{}
%IslandsEngine.Coordinate{guessed?: false, in_island: :none}
```

The dot notation gives us access to the values from the bound variable as well.

```
iex> coord.in_island
:none
```

```
iex> coord.guessed?
false
```

The dot won't let us set values, though. Elixir values are immutable, so we won't ever be able to change a value in place.

```
iex> coord.guessed? = true
** (CompileError) iex:4: cannot invoke remote function coord.guessed?/0 inside match
    (elixir) src/elixir_translator.erl:234: :elixir_translator.translate/2
    (elixir) src/elixir_clauses.erl:26: :elixir_clauses.match/3
    (elixir) src/elixir_translator.erl:18: :elixir_translator.translate/2
```

A struct is a special case of a map. It includes a __struct__ field whose value is the module name. Because of that, Map functions work with structs as well, and Map.put/3 is ideal to set new values in the struct bound to the coord variable. Map.put/3 will return an entirely new struct, and we can re-bind coord to that new value.

```
iex> coord = Map.put(coord, :guessed?, true)
%IslandsEngine.Coordinate{guessed?: true, in_island: :none}
```

When we check the value of :guessed? now, it is true.

```
iex> coord.guessed?
true
```

Having a Struct to represent our data is a fine first step, but structs live short lives. Once they pass out of scope, the garbage collector reclaims their memory, and they are gone forever. Our game will continue running in the BEAM as long as we keep playing. To keep our struct available throughout the game, we're going to need a persistent process to hold it.

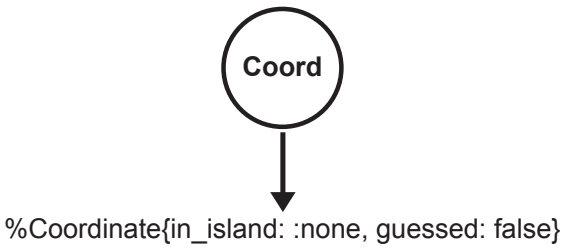
Structs are also only available within a single process, but in Islands, multiple processes will need access to the state of a coordinate. We're going to need something that can communicate with many processes as well.

Elixir Agents were designed to fulfill both of these requirements. They are long-lived processes which hold state, and they respond to requests from other processes.

Transform It Into an Agent

Agents are an abstraction built on top of GenServer, which comes with OTP. We'll cover GenServers in Chapter 3, *Wrap It Up In a GenServer*, on page 47. For now, think of Agents as simple, lightweight servers that run in the virtual machine.

Our job for this section is to turn the `IslandsEngine.Coordinate` module into an agent that holds a `%Coordinate{}` as its state.



The first step is to start up a new agent process with the state we want.

There are four functions we could use to do this. The main difference between them is whether or not the agent creates a link to the process that calls it. This will come into play when we talk about supervision trees in *the (as yet) unwritten chapter.design_for_recovery*. To link the agent, use the `Agent.start_link` function. Otherwise use `Agent.start`.

Linking Processes

When we link two processes, if one process crashes, the other will terminate as well. If we link more than two processes and one crashes, all the rest will terminate. This is an important feature that helps remove explicit error handling code in our programs. It also leads to Elixir's great fault tolerance.

Each of these two functions has two variations based on the number of arguments it takes. The two arity version takes an anonymous function to set the state, and an optional list of options. The four arity version takes a module name, a function name, and arguments as well as an optional list of options. The module, function, and arguments point to the code which will set the initial state.

We'll use the linked, two arity version here. Let's head back to the IEx session to try it out.

```
iex> Agent.start_link(fn -> %Coordinate{} end)
{:ok, #PID<0.118.0>}
```

The anonymous function we use to set the initial state takes no arguments and returns a fresh coordinate struct with default values.

Agent.start_link/2 always returns a tagged tuple, a tuple with an atom denoting the status as the first element, and the body of the return as the second. Tagged tuples are ideal for pattern matching. They tell us what's going on right on the surface of the data. There's no need to query a return to see if it was successful.

Agent.start_link/2 has three possible return values, {:ok, pid} for success, {:error, {:already_started, pid}} for when the agent PID already exists, and {:error, reason} if the Agent cannot be started.

PID is Short for Process Identifier

A PID is a first class datatype in Elixir. You can think of it as a datatype which holds the address of a process that you can send messages to.

We can pattern match for the success case, and bind the coord variable to the body of the return in one line. This will fail if there are problems starting the Agent. That's ok.

```
iex> {:ok, coord} = Agent.start_link(fn -> %Coordinate{} end)
{:ok, #PID<0.120.0>}
```

This gets the job done, but if we scan the Agent.start_link/2 call, only the coordinate struct gives us any idea what type of agent we're starting. If our initial state had been an empty list or map, we would have no clue.

We can fix this with a new function in the IslandsEngine.Coordinate module that wraps the bare Agent.start_link/2 call.

agent/lib/coordinate.ex
```
def start_link() do
  Agent.start_link(fn -> %Coordinate{} end)
end
```

Let's go try that out in IEx.

```
iex> {:ok, coordinate} = Coordinate.start_link
** (UndefinedFunctionError)
    undefined function: IslandsEngine.Coordinate.start_link/0
    (islands_engine) IslandsEngine.Coordinate.start_link()
```

Oops! We need to recompile the IslandsEngine.Coordinate module to pick up our new function.

```
iex> r(IslandsEngine.Coordinate)
lib/coordinate.ex:1: warning: redefining module IslandsEngine.Coordinate
{:reloaded, IslandsEngine.Coordinate, [IslandsEngine.Coordinate]}
```

The r/1 Function

The IEx helper function r/1 recompiles a single module, and loads the newly recompiled code into memory. It won't touch the .beam file for that module on disk.

This is different from another IEx helper function recompile/0. recompile/0 recompiles the whole project from IEx instead of a single module.

Now it should work, and we can bind the coordinate variable to the agent PID as we did before.

```
iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.106.0>}
```

Great, we started the agent, and we have its PID. Now how do we interact with its state?

Access The Agent's State

The two most basic things we should be able to do are get the state and update it. The Agent module includes functions for both, Agent.get and Agent.update. Each comes in either a three arity or five arity form. The three arity forms

take the agent PID, an anonymous function to manipulate the state, and an optional timeout value which defaults to 5000 milliseconds. The five arity forms take the agent PID, a module name, a function name, arguments, and an optional timeout value just like the three arity version.

We'll use the three arity versions for both functions.

Let's look at the simplest first, retrieving state from the agent with Agent.get/3. When evaluating this function, the agent will pass the state it holds as the argument to the anonymous function we passed in. If the anonymous function simply returns the argument, we'll get the full state back out.

Let's head back to the IEx session and give it a try.

```
iex> coord = Agent.get(coordinate, fn state -> state end)
%IslandsEngine.Coordinate{guessed?: false, in_island: :none}
```

The Missing Third Argument

You might have noticed that we called the Agent.get/3 function, but we only passed in two arguments. The third argument is a timeout which already has a default value of 5000 milliseconds. Since it has a default value, we don't need to specify it.

By changing the anonymous function, we can also get specific values back from Agent.get/3.

```
iex> guessed = Agent.get(coordinate, fn state -> state.guessed? end)
false
```

```
iex> in_island = Agent.get(coordinate, fn state -> state.in_island end)
:none
```

Again, it would be great to have named functions to get individual pieces of the agent's state. That's really easy. All we need to do is wrap Agent.get/3 calls in new functions and change the anonymous functions to return what we want.

Here's the one for :guessed?.

agent/lib/coordinate.ex

```
def guessed?(coordinate) do
  Agent.get(coordinate, fn state -> state.guessed? end)
end
```

Here's the one for :island.

agent/lib/coordinate.ex

```
def island(coordinate) do
  Agent.get(coordinate, fn state -> state.in_island end)
end
```

Ultimately, we'll need a function to determine whether a coordinate has been hit or not. The piece we're missing is a function that tells us if a coordinate is in an island.

Let's define an in_island?/1 function for this.

agent/lib/coordinate.ex
```
def in_island?(coordinate) do
  case island(coordinate)) do
    :none -> false
    _     -> true
  end
end
```

In in_island/1, we use the Coordinate.island/1 function to get the agent's state, which is a coordinate struct. Then we use the value of the :in_island key in a case statement to determine the return value.

We can use these to implement a Coordinate.hit?/1 function to return a boolean representing whether a coordinate is hit.

agent/lib/coordinate.ex
```
def hit?(coordinate) do
  in_island?(coordinate) && guessed?(coordinate)
end
```

Let's try it out in IEx, but this time, we'll go back to Agent.start_link/2 and set some new values for the initial state. Remember to recompile IslandsEngine.Coordinate first for the new functions to be available.

```
iex> {:ok, coordinate} = Agent.start_link(fn -> %Coordinate{guessed?: true,
                                                  in_island: :my_island} end)
{:ok, #PID<0.114.0>}
iex> Coordinate.guessed?(coordinate)
true
iex> Coordinate.in_island?(coordinate)
true
iex> Coordinate.hit?(coordinate)
true
```

So far so good, but to be really useful, we need to be able to change a coordinate's state to reflect changes in the game. That's up next.

Transform The Agent's State

Let's move on to updating agent state with Agent.update/2. Updates work a little differently from gets because we need to specify a key or keys to update as well as their new values. Let's update the :guessed? key to true.

```
iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.126.0>}

iex> Agent.update(coordinate, fn state -> Map.put(state, :guessed?, true) end)
:ok

iex> Coordinate.guessed?(coordinate)
true
```

A player can only guess an individual coordinate once in the course of a game. Players can never "un-guess" a coordinate. That means the value of the :guessed? key can only ever change from false to true, and it can only do that once. As before, let's wrap Agent.update/2 in a function that will take an agent PID and change the value of its :guessed? key to true.

agent/lib/coordinate.ex

```
def guess(coordinate) do
  Agent.update(coordinate, fn state -> Map.put(state, :guessed?, true) end)
end
```

Let's recompile the module and test it out.

```
iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.127.0>}

iex> Coordinate.guessed? coordinate
false

iex> Coordinate.guess coordinate
:ok

iex> Coordinate.guessed? coordinate
true
```

That's looking really nice.

Players can move their islands as often as they like before they are set, so coordinates can go in and out of islands. This means that the value of the :in_island key can potentially change multiple times. The wrapper function for this key will need to take an atom argument for the new value in addition to the agent PID. We can ensure that the new value is an atom by adding a guard clause, when is_atom value.

agent/lib/coordinate.ex

```
def set_in_island(coordinate, value) when is_atom value do
  Agent.update(coordinate, fn state -> Map.put(state, :in_island, value) end)
end
```

Let's try it out. Again, remember to recompile the module first.

```
iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.132.0>}
```

```
iex> Coordinate.in_island?(coordinate)
false

iex> Coordinate.set_in_island(coordinate, :my_island)
:ok

iex> Coordinate.in_island?(coordinate)
true

iex> Coordinate.set_in_island(coordinate, :none)
:ok

iex> Coordinate.in_island?(coordinate)
false
```

Just to make sure the guard clause works, let's pass it an incorrect value, the string ":my_island" instead of the atom.

```
iex> Coordinate.set_in_island(coordinate, ":my_island")
** (FunctionClauseError)
    no function clause matching in IslandsEngine.Coordinate.set_in_island/2
    (islands_engine)
    lib/coordinate.ex:14:
    IslandsEngine.Coordinate.set_in_island(#PID<0.132.0>, ":my_island")
```

As we had hoped, that returns a FunctionClauseError.

This is all we really need a coordinate to do, but we have an opportunity to add a little bit that will help us as we build more complex relationships.

Right now, it's easy to call a few individual functions to see parts of the coordinate's state. What would be great is if we had a single function to show the whole state as a string.

Let's write that now.

agent/lib/coordinate.ex

```
def to_string(coordinate) do
  "(in_island:#{island(coordinate)}, guessed:#{guessed?(coordinate)})"
end
```

The to_string/1 function uses Elixir's string interpolation to create a string with the coordinate struct's keys and values.

Let's see it in action.

```
iex> alias IslandsEngine.Coordinate
IslandsEngine.Coordinate

iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.154.0>}

iex> Coordinate.to_string(coordinate)
"(in_island:none, guessed:false)"
```

```
iex> Coordinate.guess(coordinate)
:ok
iex> IO.puts Coordinate.to_string(coordinate)
(in_island:none, guessed:true)
:ok
```

At the moment, the coordinate reports that it isn't in an island at all. We'll fix that shortly.

With that, our coordinate module does everything we need it to. It's time to start modeling relationships between agents.

Model a Relationship With a List

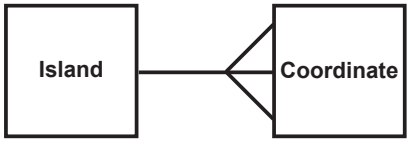
Complex data has structure and hierarchy. Pieces of data compose together to form more complex pieces. Since agents can hold any Elixir data structure as state, we'll be able to build complex relationships with agents.

Building complex relationships out of loosely coupled agents will make our domain flexible and easy to work with.

At a fundamental level, an island is a container for a group of coordinates. We can see this as a one-to-many relationship. One Island has many Coordinates.

If we were working with a database, we would need two tables—coordinates and islands. The coordinates table would need a foreign key to the islands table to define the relationship. Then we would need two entities in our domain that know how to communicate with those tables, and they would both need code to define the relationships and interactions between them.

With agents, this becomes much simpler. "One Island has many Coordinates" translates to, "An Island agent holds a list of Coordinate PIDs in its state."



Like coordinates, islands need to exist for as long as the game continues, so we'll use an Agent to hold the data.

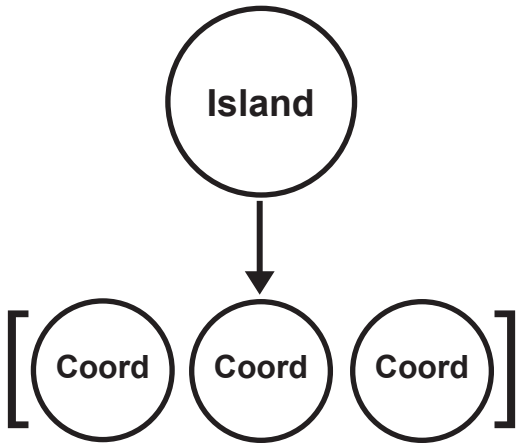
At the beginning of the game, players will not have placed their islands on the board, so the islands won't have any coordinates. When a player does

place an island on the board, the game will need to associate the coordinates the island occupies on the board with the island. If a player moves an island, the game will need to disassociate it from the coordinates it used to have, and re-associate it with the new ones it lands on.

Composing the relationship between coordinates and islands with a simple list removes the need for extra entities in the system.

We won't need a struct or a map to model island data because we don't need to know anything specific about an island itself. The initial state will be an empty list, and that's what we'll use to start a new agent.

Besides keeping track of its coordinates, an Island needs to be able to tell us whether or not it is forested. An island can decide whether it is forested by asking each of it's coordinates if it is hit. If they are all hit, the island is forested, otherwise not.



We'll need a new file in the lib directory called island.ex to define the Island module.

```
defmodule IslandsEngine.Island do
  alias IslandsEngine.Coordinate

end
```

The alias for the IslandsEngine.Coordinate module will come in handy later on.

We'll need an Island.start_link/0 function to wrap Agent.start_link/2, just as in the Coordinate module. Island agents will have an empty list as initial state.

agent/lib/island.ex
```
def start_link() do
  Agent.start_link(fn -> [] end)
end
```

Let's just check to make sure that works.

```
iex> alias IslandsEngine.Island
nil

iex> {:ok, island} = Island.start_link
{:ok, #PID<0.107.0>}

iex> Agent.get(island, fn state -> state end)
[]
```

That's great. A new island comes with an empty list of coordinates by default.

We'll need to be able to reset an island's list of coordinates, so let's start there. The simplest and least error-prone way to implement this is to replace the original list with a new one.

We know that Agent.update/3 let's us change an agent's state. We can use that to implement an IslandsEngine.Island.replace_coordinates/2 function that takes a list of coordinates and makes that the new state. While we're at it, let's add a guard clause that ensures the new coordinates will be a list.

agent/lib/island.ex
```
def replace_coordinates(island, new_coordinates) when is_list new_coordinates do
  Agent.update(island, fn _state -> new_coordinates end)
end
```

```
iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.116.0>}

iex> Island.replace_coordinates(island, [coordinate])
:ok

iex> Agent.get(island, fn state -> state end)
[#PID<0.116.0>]
```

To make sure that the guard clause works, let's try calling IslandsEngine.Island.replace_coordinates/2 with a single coordinate.

```
iex> coordinate
#PID<0.116.0>

iex> Island.replace_coordinates(island, coordinate)
** (FunctionClauseError)
    no function clause matching in IslandsEngine.Island.replace_coordinates/2
    (islands_engine)
    lib/island.ex:9:
    IslandsEngine.Island.replace_coordinates(#PID<0.107.0>, #PID<0.116.0>)
```

That's just what we want. Elixir raises a FunctionClauseError when we only pass in a single coordinate.

Now let's figure out if an island is forested.

Elixir's Enum.all?/2 allows us to apply an anonymous function to each element of a list. Enum.all?/2 returns true if the anonymous function returns true for all elements. Otherwise it returns false.

We can craft a function that will get the list of coordinates from the agent, and then use Coordinate.hit?/1 in the anonymous function to check whether or not each coordinate has been hit.

agent/lib/island.ex

```
def forested?(island) do
  island
  |> Agent.get(fn state -> state end)
  |> Enum.all?(fn coord -> Coordinate.hit?(coord) end)
end
```

That looks like it should work. Let's give it a try in the console.

```
iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.131.0>}

iex> Island.replace_coordinates(island, [coordinate])
:ok

iex> Coordinate.set_in_island(coordinate, :my_island)
:ok

iex> Island.forested?(island)
false

iex> Coordinate.guess(coordinate)
:ok

iex> Island.forested?(island)
true
```

That's exactly what we want.

As we build a more complex structure with more relationships, it will be useful to have a way to represent an island as a string. This is very similar to what we did with coordinates. In fact, we'll make good use of the Coordinate.to_string/1 function we already have.

The difference is that we'll use a helper function to enumerate over the coordinates, get their string representations, and join those strings together.

agent/lib/island.ex

```
def to_string(island) do
  "[" <> coordinate_strings(island) <> "]"
```

```
end

defp coordinate_strings(island) do
  island
  |> Agent.get(fn state -> state end)
  |> Enum.map(fn coord -> Coordinate.to_string(coord) end)
  |> Enum.join(", ")
end
```

Let's see it in action.

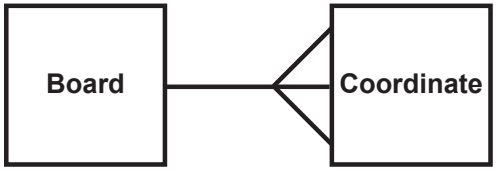
```
iex> alias IslandsEngine.Coordinate
IslandsEngine.Coordinate

iex> alias IslandsEngine.Island
IslandsEngine.Island

iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.141.0>}

iex> {:ok, island} = Island.start_link
{:ok, #PID<0.154.0>}

iex> Island.replace_coordinates(island, [coordinate])
:ok

iex> Island.to_string(island)
"[(in_island:none, guessed:false)]"

iex> IO.puts Island.to_string(island)
[(in_island:none, guessed:false)]
:ok
```

Now that we can model one-to-many relationships, let's move on to something just a little more complex.

Model a Relationship With a Map

Modeling relationships with a list is fine for anonymous access to the list elements. There are times, though, when we need to access elements by name. That's where maps come in.

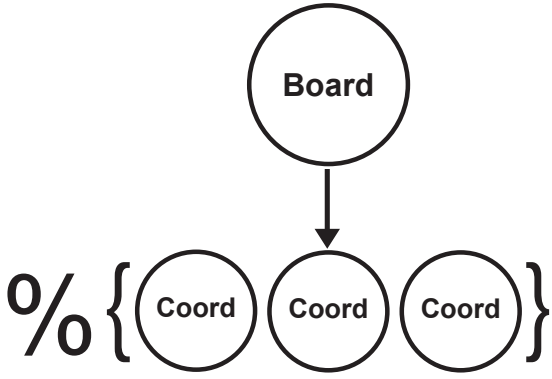
In Islands, a Board needs to act as an organizer. A Board is a grid of a hundred named boxes. We'll hold the state of each named box in a Coordinate. We'll need to model the Board in such a way that we can reference each Coordinate by the letter-number name of the box they represent so that we can read and manipulate the state of any box. A map is the perfect choice for modeling this relationship between a board and its coordinates.



Boards have many coordinates and a coordinate belong to a board. If we were modeling this relationship for a database, the coordinates table would have a foreign key to the board table. It would also contain a column for the coordinate name—"a7" or "g10". The name is not necessary for Coordinates in relation to Islands. It only makes sense in the context of a Board.

We don't need to make each coordinate store its name when only the Board needs it. We can keep Coordinates slim and only add the name in the context that it's needed, the Board. We'll use a map to model this, and the key will be the letter-number name.

We could use a struct here, but in order to really use it as a struct, and differentiate it from a map, we would need to list all one hundred coordinate names as keys in the struct definition followed by a default value. No thanks. We'll use a map instead and generate the keys and values on the fly when we start up the board.



We'll need a file in the lib directory for the IslandsEngine.Board module, lib/board.ex. Let's prefix the module name with IslandsEngine as we did with Coordinates and Islands.

agent/lib/board.ex
```
defmodule IslandsEngine.Board do
```

```
  alias IslandsEngine.Coordinate
```

It will need a Board.start_link/0 function which calls Agent.start_link/2.

```
def start_link do
  Agent.start_link(fn -> %{} end)
end
```

Note that we used an empty map as the initial state. This will get us started, but an empty board doesn't really get us too far. We could really use a fully populated board with all the keys and coordinate agents already in place.

To accomplish this, we'll need a function that generates a list of all the keys in a board. That will be all the combinations of the letters a through j, and all the numbers 1 through 10. Along the way, we'll turn those keys into atoms.

For convenience, let's set those two lists as module attributes in the Board module.

agent/lib/board.ex

```
@letters ~W(a b c d e f g h i j)
@numbers [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

First up is the function to generate a list of all the keys.

agent/lib/board.ex

```
defp keys() do
  for letter <- @letters, number <- @numbers do
    String.to_atom("#{letter}#{number}")
  end
end
```

This uses two "for" comprehensions. It's a common pattern with some similarities to nested "for" loops in other languages. Comprehensions pull elements from a list one at a time and execute a block with that value. Nesting the comprehensions as we have will ensure that we'll generate all the possible combinations of the elements from both lists for the block.

Next we'll need a function to enumerate over the list of keys, start a coordinate for each key, and set the PID of that coordinate as the value for the key in the map.

agent/lib/board.ex

```
defp initialized_board() do
  Enum.reduce(keys(), %{}, fn(key, board) ->
    {:ok, coord} = Coordinate.start_link
    Map.put_new(board, key, coord)
  end )
end
```

We're using Enum.reduce/3 in the initialization function because we need to put all the keys into the same map. If we used Enum.each/2 or Enum.map/2, they would return a list of maps with one key/value each—clearly not what we want. By generating all the keys as a list first, then reducing over those, we get a single map with all the keys and values set.

Now we can use Board.initialized_board/0 to set the initial state in our Board.start_link/0 function. This will give us a fully initialized board every time we start a new board agent.

agent/lib/board.ex
```
def start_link() do
  Agent.start_link(fn -> initialized_board() end)
end
```

The board will provide a way for the rest of the game to interact with individual Coordinates by name. This makes a board the de facto interface for individual Coordinates. This means we'll need to add public functions to the Board module for all those actions.

All of these function will have a common pattern. They'll need to get an individual coordinate PID from the board, and call one of the Coordinate Agent functions on it. Let's write a function that will do the first part of that pattern for us—getting a coordinate from the board by name.

agent/lib/board.ex
```
def get_coordinate(board, key) when is_atom key do
  Agent.get(board, fn board -> board[key] end)
end
```

Now that we have the function to get coordinates by name, we can easily implement the rest.

A big part of the game is guessing coordinates, and here's a function to do that.

agent/lib/board.ex
```
def guess_coordinate(board, key) do
  get_coordinate(board, key)
  |> Coordinate.guess
end
```

It just gets the coordinate PID by name from the board's state, and calls Coordinate.guess/1 with it.

Once we can guess coordinates, we'll need to know whether they have been hit or not. Board.coordinate_hit?/2 follows the same pattern as Board.guess_coordinate/2

agent/lib/board.ex
```
def coordinate_hit?(board, key) do
  get_coordinate(board, key)
  |> Coordinate.hit?
end
```

We'll need to be able to set a coordinate in an island.

agent/lib/board.ex
```
def set_coordinate_in_island(board, key, island) do
  get_coordinate(board, key)
  |> Coordinate.set_in_island(island)
end
```

Finally, we'll need to know which island a coordinate is in, if any.

agent/lib/board.ex
```
def coordinate_island(board, key) do
  get_coordinate(board, key)
  |> Coordinate.island
end
```

With that, Boards do what we need them to do. Let's see how those functions work in the console.

First, we'll start a board and get a coordinate from it.

```
iex> alias IslandsEngine.{Board, Coordinate}
[IslandsEngine.Board, IslandsEngine.Coordinate]

iex> {:ok, board} = Board.start_link
{:ok, #PID<0.129.0>}

iex> coordinate = Board.get_coordinate(board, :j10)
#PID<0.229.0>
```

Now let's try out guessing a coordinate.

```
iex> IO.puts Coordinate.to_string(coordinate)
(in_island:none, guessed:false)
:ok

iex> Board.guess_coordinate(board, :j10)
:ok

iex> IO.puts Coordinate.to_string(coordinate)
(in_island:none, guessed:true)
:ok
```

That looks great. Now let's try out setting a coordinate in an island.

```
iex> Board.set_coordinate_in_island(board, :j10, :new_island)
:ok

iex> IO.puts Coordinate.to_string(coordinate)
```

```
(in_island:new_island, guessed:true)
:ok

iex> Board.coordinate_island(board, :j10)
:new_island
```

That all looks like just what we want.

As we've done for Coordinates and Islands, we'll need a function for a string representation of a Board.

agent/lib/board.ex

```
def to_string(board) do
  "%{" <> string_body(board) <> "}"
end

defp string_body(board) do
  Enum.reduce(keys(), "", fn key, acc ->
    coord = get_coordinate(board, key))
    acc <> "#{key} => #{Coordinate.to_string(coord)},\n"
  end)
end
```

Board.to_string/1 enumerates over all the keys, and calls Coordinate.to_string/1 on each coordinate, concatenating all the resulting strings together.

```
iex> alias IslandsEngine.Board
IslandsEngine.Board

iex> {:ok, board} = Board.start_link
{:ok, #PID<0.129.0>}

iex> Board.to_string(board)
"%{a1 => (in_island:none, guessed:false),\na2 => (in_island:none, guessed:false),\n
   a3 => (in_island:none, guessed:false),\na4 => (in_island:none, guessed:false),\n
. . .
guessed:false),\nj10 => (in_island:none, guessed:false),\n}"
```

That big string is pretty hard to read, but if we pass it into IO.puts/1, the newline characters will do their job and make it a little easier for us.

```
iex(6)> IO.puts Board.to_string(board)
%{a1 => (in_island:none, guessed:false),
a2 => (in_island:none, guessed:false),
a3 => (in_island:none, guessed:false),
a4 => (in_island:none, guessed:false),
a5 => (in_island:none, guessed:false),
a6 => (in_island:none, guessed:false),
a7 => (in_island:none, guessed:false),
a8 => (in_island:none, guessed:false),
a9 => (in_island:none, guessed:false),
a10 => (in_island:none, guessed:false),
b1 => (in_island:none, guessed:false),
```

```
. . .
}
:ok
```

The board is shaping up nicely. It does all we need it to. We haven't experimented with any of its functions in IEx so far. Don't worry, we'll get our chance in a bit.

Now, let's add another type of data structure to our relationship modeling.

Model Relationships With a Struct

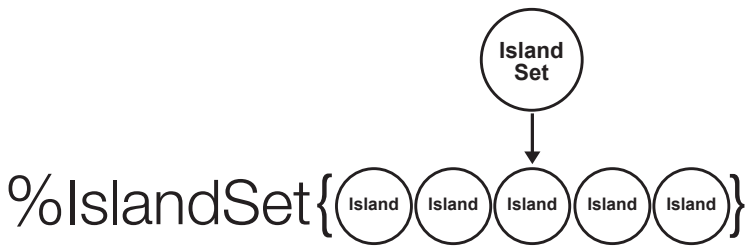
Sometimes we can be even more specific about the relationship we're modeling. When we have a small number of child elements that we know by name, and that we know in advance, we can use a struct to hold that state and compose the relationship.

The game requires two players, and that each player will have a board and a set of islands. That sentence gives us all the info we'll need to move forward. It points out that there are two entities that we haven't modeled yet, an island set and a player.

Island Sets

Let's start with island sets. We'll see how we can ensure that both players have exactly the same number and type of islands.

An island set will have one each of five different island shapes—an atoll, a dot, an "L" shape, an "S" shape, and a square. We know up front that each set will have these and only these islands, so a struct is the right data structure to model them.



Let's create a new file for our IslandSet module, lib/island_set.ex. We know we'll be storing islands here, so let's alias IslandsEngine.Island. We also know we'll need a struct, so let's alias IslandsEngine.IslandSet as well.

```
defmodule IslandsEngine.IslandSet do
  alias IslandsEngine.{Island, IslandSet}
end
```

Now we need to define the struct. It will need a key for each island type.

agent/lib/island_set.ex
```
defstruct atoll: :none, dot: :none, l_shape: :none, s_shape: :none, square: :none
```

Elixir defines structs at compile time, but we start agents at runtime. That means we can't use island agents as default values. We'll use :none for the defaults and write a function to swap in new island agents when needed.

As with all the agents we've defined so far, we'll need a function to start a new island set agent. Let's write that now.

```
def start_link() do
  Agent.start_link(fn -> %IslandSet{} end)
end
```

That will set an empty %IslandSet{} struct as the agent's state, which will work but isn't ideal. It would be best if we could set a fully initialized struct as the agent's state instead.

We'll tackle this just as we did in the Board module. The initial state will be the return value of a function that initializes an IslandSet struct automatically.

agent/lib/island_set.ex
```
def start_link() do
  Agent.start_link(fn -> initialized_set() end)
end
```

The initialized_set/1 function looks a lot like the initialization function we wrote for the Board module as well.

agent/lib/island_set.ex
```
defp initialized_set() do
  Enum.reduce(keys(), %IslandSet{}, fn key, set ->
    {:ok, island} = Island.start_link
    Map.put(set, key, island)
  end)
end
```

We use an IslandSet struct as the accumulator. The same struct will come into each enumeration as an argument, the anonymous function will set a new Island agent on it, and then Enum.reduce/3 will return it to act as an argument for the next enumeration. The fully initialized struct will be the final return value.

Enum Clarification

We just said that we were passing the same struct into each enumeration. That's not completely true. In each enumeration, as we set a new value in the struct, we're really creating a copy of the old struct with the new value included in the copy. It may look like we're updating the island set in place, but that's not what actually happens.

You may have noticed that we're calling our own function to get the struct keys instead of using the built in Map.keys/1 function. First, let's go ahead and add that function.

agent/lib/island_set.ex

```
defp keys() do
  %IslandSet{}
  |> Map.from_struct
  |> Map.keys
end
```

Now, let's see why that was necessary.

Open up a console session and alias IslandsEngine.IslandSet.

```
iex> alias IslandsEngine.IslandSet
nil
```

```
iex> island_set = %IslandSet{}
%IslandsEngine.IslandSet{atoll: nil, dot: nil, l_shape: nil, s_shape: nil,
 square: nil}
```

```
iex> Map.keys(island_set)
[:__struct__, :atoll, :dot, :l, :s, :square]
```

Map.keys/1 returns all the keys of the struct, including :__struct__. If we used this collection in Enum.reduce/3, it would reset the value of :__struct__ from the module name to an island agent PID. That would cause real problems if we ever tried to pattern match on the struct.

Our new keys/1 function fixes that by generating a new map from the struct, eliminating the :__struct__ key. Values in Elixir are immutable, so our original island_set remains unchanged.

Let's hop back into the console to check our work.

```
iex> alias IslandsEngine.IslandSet
IslandsEngine.IslandSet
```

```
iex> {:ok, island_set} = IslandSet.start_link
{:ok, #PID<0.112.0>}
```

```
iex> island_set_state = Agent.get(island_set, &(&1))
```

```
%IslandsEngine.IslandSet{atoll: #PID<0.113.0>, dot: #PID<0.114.0>,
 l_shape: #PID<0.115.0>, s_shape: #PID<0.116.0>, square: #PID<0.117.0>}
```

A New Construct

We used a new Elixir construct here, &(&1). The outer part, &(), is the function capture operator. It creates an anonymous function. The &1 part is the first argument to the function, and we're simply returning it. &(&1) is equivalent to the fn state -> state end that we have been using.

That looks good. Now let's check the state of the atoll island.

```
iex> Agent.get(island_set_state.atoll, &(&1))
[]
```

That's what we want, an empty list of coordinates.

Just as we've done with all the agents so far, let's create a string representation of an island set. As we did with boards, we'll split this into two functions. One will generate the body of the string by enumerating over the keys and getting a string representation of each island, which in turn generates a string representation of each coordinate in the island.

The public function just puts the front and back ends on that body.

agent/lib/island_set.ex

```
def to_string(island_set) do
  "%IslandSet{" <> string_body(island_set) <> "}"
end

defp string_body(island_set) do
  Enum.reduce(keys(), "", fn key, acc ->
    island = Agent.get(island_set, &(Map.fetch!(&1, key)))
    acc <> "#{key} => " <> Island.to_string(island) <> "\n"
  end)
end
```

Let's see how this works in the console.

```
iex> alias IslandsEngine.IslandSet
IslandsEngine.IslandSet

iex> {:ok, island_set} = IslandSet.start_link
{:ok, #PID<0.245.0>}

iex> IslandSet.to_string(island_set)
"%IslandSet{atoll => []\ndot => []\n
            l_shape => []\ns_shape => []\nsquare => []\n}"

iex> IO.puts IslandSet.to_string(island_set)
%IslandSet{atoll => []
dot => []
```

```
l_shape => []
s_shape => []
square => []
}
:ok
```

Great, that's what we would expect from a brand new island set. Now let's see what happens when we set a coordinate in one of the islands.

```
iex> alias IslandsEngine.{Coordinate, Island}
[IslandsEngine.Coordinate, IslandsEngine.Island]

iex> dot = Agent.get(island_set, fn state -> state.dot end)
#PID<0.247.0>

iex> {:ok, coordinate} = Coordinate.start_link
{:ok, #PID<0.255.0>}

iex> Island.replace_coordinates(dot, [coordinate])
:ok

iex> Coordinate.set_in_island(coordinate, :dot)
:ok

iex> IO.puts IslandSet.to_string(island_set)
%IslandSet{atoll => []
dot => [(in_island:dot, guessed:false)]
l_shape => []
s_shape => []
square => []
}
:ok
```

That's great. The string representation of the dot island shows a list with a single coordinate that is in the dot island and hasn't been guessed.

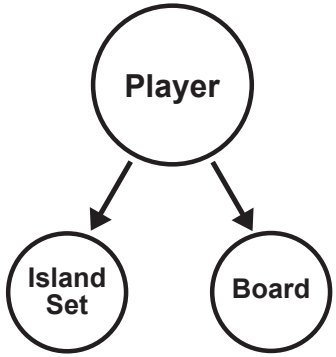
For now, that's all we'll need from the IslandSet module.

Let's move on to the last agent we'll be modeling with a struct, a player.

Players

We need to give players this same treatment: define the state we need and make sure it's properly set when we start the agent.

A player has a board and a set of islands. A player should also have a name to display on screen. Those are the three keys we'll use to create a player struct—:name, :board, and :island_set.



We'll need a new file at lib/player.ex to define the IslandsEngine.Player module. Let's alias the IslandsEngine.Board, IslandsEngine.Coordinate, IslandsEngine.IslandSet, and IslandsEngine.Player modules while we're here.

```
defmodule IslandsEngine.Player do
  alias IslandsEngine.{Board, Coordinate, IslandSet, Player}
end
```

Based on what we just said about the keys, here's what the player struct should look like.

agent/lib/player.ex

```
defstruct name: :none, board: :none, island_set: :none
```

Again, all the default values are :none.

We'll need a start_link/0 function, as we've defined with all the others. The key will be to start a new board and a new island set, then set the PIDs of those processes as the values for their respective keys.

agent/lib/player.ex

```
def start_link(name \\ :none) do
  {:ok, board} = Board.start_link
  {:ok, island_set} = IslandSet.start_link
  Agent.start_link(fn -> %Player{board: board, island_set: island_set, name: name} end)
end
```

Notice that we did not set a value for the :name key in the start_link/0 function. We may not know a player's name when we start the agent. Instead, let's write a function for setting a player's name when we need to.

agent/lib/player.ex

```
def set_name(player, name) do
  Agent.update(player, fn state -> Map.put(state, :name, name) end)
end
```

As we have with coordinates and islands, let's build a string representation of a player to help us visualize a player's state as we work. The sigil ~s() will allow us to use double quotes inside of a string so that players' names will appear as quoted strings. We'll also rely on the to_string/1 functions of both the IslandSet and Board modules to help us out.

agent/lib/player.ex

```
def to_string(player) do
  "%Player{" <> string_body(player) <> "}"
end

defp string_body(player) do
  state = Agent.get(player, &(&1))
  ":name => " <> name_to_string(state.name) <> ",\n" <>
  ":island_set => " <> IslandSet.to_string(state.island_set) <> ",\n" <>
  ":board => " <> Board.to_string(state.board)
end

defp name_to_string(:none), do: ":none"
defp name_to_string(name), do: ~s("#{name}")
```

Let's take the Player out for a spin in the console to see how it behaves.

We'll need to alias IslandsEngine.Player and then bind the player variable to a new Player struct. Once we have that, we can initialize the struct and see what we get.

```
iex> alias IslandsEngine.Player
IslandsEngine.Player

iex> {:ok, player} = Player.start_link
{:ok, #PID<0.236.0>}

iex> IO.puts Player.to_string(player)
%Player{:name => :none,
:island_set => %IslandSet{atoll => []
dot => []
l_shape => []
s_shape => []
square => []
},
:board => %{a1 => (in_island:none, guessed:false),
a2 => (in_island:none, guessed:false),
. . .
j10 => (in_island:none, guessed:false),
}}
:ok
```

This looks like exactly what we want. The player has a Board agent as well as an IslandSet full of Island agents.

Now let's see the set_name/2 function in action.

```
iex> Player.set_name(player, "Adrian")
:ok
iex> IO.puts Player.to_string(player)
%Player{:name => "Adrian",
:island_set => %IslandSet{atoll => []
dot => []
l_shape => []
s_shape => []
square => []
},
:board => %{a1 => (in_island:none, guessed:false),
a2 => (in_island:none, guessed:false),
. . .
j10 => (in_island:none, guessed:false),
}}
:ok
```

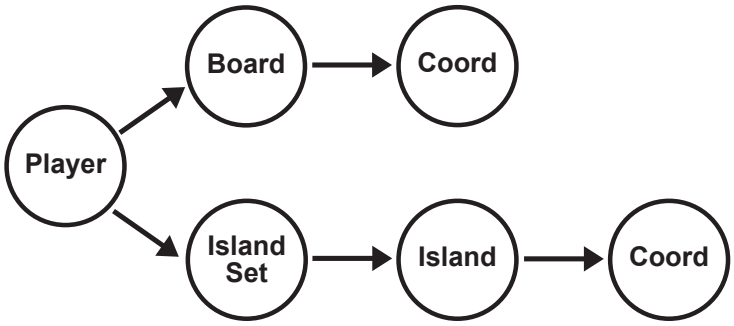
That looks great. It set the player's name to "Adrian" as we expected.

Now that we have all the pieces in place to create players for Islands, let's take a look at how this helps us.

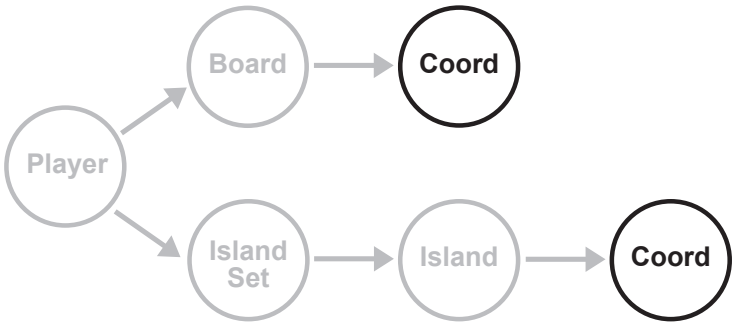
A Single Representation

Earlier in this chapter, we talked about the importance of having a single representation for each piece of state in the system. Now that we've built up the relationships between all of our agents, we can see this first hand.

We've built up a tree structure of agents with a player at the root. One path goes through a board to a map of coordinates. The other goes through an island set through a struct of islands to a list of coordinates. Here's a simplified diagram of that hierarchy.



The interesting thing is that the leaves of both branches of this tree are always coordinates. In other words, any coordinate can simultaneously be part of the board and part of an island. The challenge with a structure like this is keeping coordinate data in synch on both branches of the tree.



If we had chosen to model this tree as a huge, single data structure, every time we updated coordinate data on one branch, we would need to go update it on the other as well. That's a recipe for bugs if ever there was one.

Instead, we broke the huge structure into manageable pieces, and we stored those pieces in agents that can communicate and coordinate. The agent PIDs we use as the values inside agent state are simply addresses pointing to the source.

Let's take a look at how this setup completely eliminates the problem of coordinating state updates.

In an IEx session, let's alias the Board, Coordinate, and Island modules. Then let's start a board and an island.

```
iex> alias IslandsEngine.{Board, Coordinate, Island}
[IslandsEngine.Board, IslandsEngine.Coordinate, IslandsEngine.Island]

iex> {:ok, board} = Board.start_link
{:ok, #PID<0.112.0>}

iex> {:ok, island} = Island.start_link
{:ok, #PID<0.214.0>}
```

Next, let's get the PID for the first coordinate in the board, the value for the :a1 key, and set it in the island.

```
iex> a1 = Board.get_coordinate(board, :a1)
#PID<0.113.0>

iex> Island.replace_coordinates(island, [a1])
:ok
```

Once we have that, let's take a look at the string representation of the :a1 coordinate in both the board and the island.

```
iex> IO.puts Board.to_string(board)
%{a1 => (in_island:none, guessed:false),
. . .
}
:ok

iex> IO.puts Island.to_string(island)
[(in_island:none, guessed:false)]
:ok
```

They're exactly the same, which is what we would expect.

This is where things get interesting. Let's set the :a1 coordinate in :my_island and take a look at it from the perspective of both the board and the island.

```
iex> Board.set_coordinate_in_island(board, :a1, :my_island)
:ok

iex> IO.puts Board.to_string(board)
%{a1 => (in_island:my_island, guessed:false),
. . .
}
:ok

iex> IO.puts Island.to_string(island)
[(in_island:my_island, guessed:false)]
:ok
```

That's fantastic. The change we made to the coordinate through the board was automatically reflected in the island. We did zero extra work to make that happen.

Let's see that again by guessing a coordinate.

```
iex> Board.guess_coordinate(board, :a1)
:ok

iex> IO.puts Board.to_string(board)
%{a1 => (in_island:my_island, guessed:true),
. . .
}
:ok

iex> IO.puts Island.to_string(island)
[(in_island:my_island, guessed:true)]
:ok
```

As expected, we see the guess reflected in both the board and the island automatically.

This is a good foundation. The pieces are fitting together nicely already, and their interactions will become even smoother in the next chapter.

Wrapping Up

We've made good progress so far. We modeled the most important building blocks of the game. We can see how coordinates compose into islands and boards. We composed them further into island sets and players.

Our domain is simpler because of the approach we've taken. Our domain entities live single lives in our application instead of double lives in both the application and the database. Updating state means calling functions on agents instead of updating some existing state in memory and then duplicating that update in the database. And we've cut down on a whole lot of latency by not going back and forth across the network to the database.

While we are in a good position, the problem is that the application we currently have doesn't look much like a game yet. It's more of a box of pieces at this point. We'll take care of this in the next chapter when we build a GenServer to bring all the pieces together into a coherent whole.

What we'll do in this chapter

- *implement a GenServer for the game logic and public interface*
- *practice common patterns in OTP Behaviours*
- *initialize Agents and GenServers with the correct state*
- *manage complexity with composition and pipelines*
- *use process registration to name individual games*

CHAPTER 3

Wrap It Up In a GenServer

Let's face it, applications just naturally grow, and often they grow too large and complex to easily manage. The impulse to break large applications up into smaller, independent ones has been around a long time. Micro-services are currently the most popular approach for this, but long before micro-services, there was SOA, service oriented architecture.

The benefits of services are tantalizing and clear. By breaking applications apart, each new piece can focus more directly on it's given task. We can scale applications more efficiently by giving extra resources to only the services that need it.

Yet anybody who has implemented services knows it's harder than it looks. Everything becomes more complex, from setting up a development environment to deploying a new feature. The number of things that can go wrong grows enormously, and handling those failures requires careful thought and extra work.

OTP has an answer for these problems called GenServer, short for "generic server". GenServer processes give us many of the benefits of independent services without the headaches.

As we work through this chapter, we'll build a GenServer from the ground up. We'll see how to spawn new instances of this GenServer as separate Elixir processes, and we'll see how to customize its behavior to fit our needs.

First, let's take a closer look at services.

Services, A Fuller Picture

Soon after we begin programming, we're taught to break large pieces of software down into smaller ones. If a method is too large, we break it into smaller

methods with intention revealing names. Then we compose those pieces back together again to recreate the same functionality. We do the same with classes and modules.

This helps because we don't need to keep the full complexity of the larger whole in our minds at once. We can focus on the re-composed whole if we want the big picture. Good names make the overall behavior clear. If we need more detail, we can focus on any individual piece.

Services are an extension of that same impulse, scaled up.

Typically, services are completely independent of one another. They run on different machines, and they store their own data separately from other services. Yet in order for the full application to work, services need to communicate and coordinate. They need to compose back together to provide the same functionality as before, just as methods, functions, classes and modules need to.

The key to that recomposition is communication. Since they are no longer on the same machine, services need an external mechanism to talk to each other. Most often, that is an HTTP request from one service to another, but it can also be a message queue like RabbitMQ or ZeroMQ.

This separation—and need for communication—is exactly the source of all the benefits as well as all the problems associated with breaking an application up into services.

The Advantages

The forces that push developers to reach for services are direct and easy to enumerate: the need for focus, encapsulation, and scale.

Software that does one thing and does it well has been a goal since the birth of Unix. Code that is smaller, more contained, and more easily kept in a single brain is always preferable. It's easier to write and maintain. Well named, focused services show us where to start looking if something goes wrong. They point to where new code belongs when we need to add functionality.

Well designed services hide all their data and internal functionality, only exposing a public interface. The only way other services can query or manipulate that data is through the public API. By hiding the internal implementation, we can change it without breaking interactions with other services—so long as we keep the arguments and return values the same.

Different parts of applications can have very different resource needs. Some serve fewer requests, some more. Some spend a lot of resources fulfilling

requests, some much fewer. With services, we can target extra resources toward only those parts of an application that need them and spend our money more efficiently.

A Quick Example

Let's think about something concrete so we have somewhere to hang these ideas.

Imagine we're writing a very simple micro-blogging application. We need to let users create accounts, and we need to authenticate those users with a username and password. Users should be able to follow one another. They need to be able to post small messages, and they need to be able to see a timeline of messages posted by the users they follow.

We can guess that handling users and authentication is going to be much less resource intensive than showing timelines. Users only create accounts once, and they may log in only occasionally.

Finding all the posts a user should see is going to require a lot more processing—given a list other users an individual follows, we'll need to find all the posts those other users have made and put them in order in a list. We'll need to do this each time anyone views their timeline.

We can also guess that the data needs for handling users will be much different from handling users following each other. For persisting users, a simple relational database table will do, but the data for modeling users following other users is shaped more like a graph.

We could envision breaking this application up into three services: one for users, one to handle users following each other, and one for micro-posts. When a user creates and account or logs in, they would be working with the user service. When they want to follow other users, that would go through the following service. When they want to see their timeline, that would go through the post service, which would call the following service to see who the given user follows, and then create the timeline based on that list.

A production-grade application might differ to a certain degree, but to illustrate service separation, this example still holds.

The Pain Points

We don't get the benefits of services for free. Getting independent services to act in concert is challenging. Services add complexity when setting up a development environment, while testing, and during deployment. Handling

failure in dependent services when another service crashes or loses communication over the network takes extra thought, preparation, and work.

Bringing up a development environment in which several services have to be running and coordinated is non-trivial. If they communicate over HTTP, they can't all share the same port. Each service needs to know the port of all the others in order to work properly. If they use an external message queue, that has to be configured and running as well.

This is brittle. Any change might break the system and send us into a debugging cycle. It's also resource intensive because we typically need to start multiple services locally to get the application working.

Testing can be a challenge. When services interact, we need to choose between spinning up all the services to run tests, or to use mock services to maintain isolation. Using live services adds just as much complexity as it does for a development environment. If we use mocks, keeping them in sync with the services they represent will be a continuous chore.

Deployment becomes more complex as well. Managing version changes across multiple services requires careful planning. Handling breaking changes to one service means that we'll need to update others that depend on it. Then there's a little deployment dance where we need to have new code on multiple services at exactly the same time for the whole system to work.

We also need a new strategy for cross-service fault tolerance—how we keep other services up and running well when one of the services is down or misbehaving. These things can give you nightmares. At the very least, they represent considerable extra work up front in order to reap the benefits later on.

Decisions Decisions

One of the most difficult parts of implementing services is deciding where to split a large application up—determining where the facets are.

When we're talking about splitting up methods, functions, classes, and modules, we have a name for it: refactoring. Many books on this single topic exist for all sorts of languages.

We're not yet that far along as an industry with regard to services. One of the reasons is that full applications are much more complex than even classes or modules. It's just plain harder to come up with generalized rules for such complex interactions.

Another reason is that few languages have any organizing constructs between a class/module and a full application. There's no intermediary box in most

languages to group multiple classes/modules in and put a name on. Yes, there are libraries and packages, but we really don't consider them as part of an application. They're external code that brings functionality in.

Erlang developers came up with answers to these problems a couple of decades ago, and they put them in OTP. Let's take a closer look.

OTP Solutions

One of the answers OTP provides is GenServer, which allows us to create separate Elixir processes that act as servers. The other is OTP Applications, which are larger in scale. They're analogous to libraries in other languages, but we can compose them together to make larger applications in ways that we can't with regular libraries.

GenServer

GenServer processes provide most of what we want from services, and they address the problems that services create as well. They are separate Elixir processes that listen for and respond to messages from other processes. They can hold state as well as take action in the system.

Because they are separate processes that share nothing with other processes, we get the isolation and encapsulation that we're looking for. We can spawn new processes to address scaling needs just as we did with agents in the last chapter, and do it at a very granular level.

Elixir applications that use GenServer processes are just normal Elixir applications. There's no extra configuration necessary to setup a development environment. We won't need any external means to allow a GenServer process to communicate with the rest of the application.

Testing works exactly the same as with any other Elixir app. There's no need to mock another service because the application remains a single whole.

Deployments are the same as well. Whatever deployment strategy you currently use will just work. Since the GenServer is integrated with the rest of the application, there's no way to create a version mismatch.

We don't need extra planning or work to ensure fault tolerance. GenServer processes can be supervised, so we naturally get a level of fault tolerance that's very difficult to match in any other system.

OTP Applications

OTP Applications take on the "where to divide a big application" question. They are that intermediary organizing construct that most languages lack.

The first thing we need to clear up is the name. An OTP Application is not what we usually think of as a complete software application. It's a bit closer to what we would call a library or a package in other ecosystems. They are a little different, though. OTP Applications are supervised units of code that start and stop as a single entity.

We can use them as libraries or packages if we want, but they can also be integral, named, delineated parts of an application. They naturally define facets we can break an application apart by if our needs demand it.

The good news is that we've been building an OTP Application from the beginning. When we created our IslandsEngine Elixir app, the generator created it as an OTP Application.

Here's the /lib/islands_engine/application.ex file that Elixir generated for us.

```
defmodule IslandsEngine.Application do
  use Application
. . .
end
```

We're not going to deal with OTP Applications directly in this chapter. We'll see them in much more detail when we talk about Supervisors in *the (as yet) unwritten chapter.design_for_recovery*, and again when we introduce Phoenix in part two of the book.

Both GenServers and Applications are examples of what OTP calls a Behaviour. Let's take a look at those next.

Introducing OTP Behaviours

OTP is Erlang's extended standard library. It includes a number of software tools and a set of design patterns. Together, these are a treasure trove of collected wisdom and best practices for building concurrent and fault tolerant programs in Elixir and Erlang.

OTP stands for Open Telecom Platform. The name reflects Erlang's origin in the telecom industry at Ericsson. In practice, OTP is much more general than the name suggests. We can build any type of application with OTP.

OTP's toolset is extensive. It includes an in-memory key-value store (ETS), a relational database (Mnesia), monitoring and debugging tools (Observer,

Debugger), a release management tool (Reltool), a static analysis tool (Dialyzer), and that list just begins to scratch the surface. We won't cover these here, but it's good to know that they are there if you need them.

For the next few chapters, we're going to see a lot of OTP *Behaviours*. Coming up with a concise explanation of what OTP Behaviours are is difficult. In concrete terms, they are modules in OTP as well as modules that we define in our applications. But they are also design patterns that reflect best practices. The rest of this section should clear things up.

Behaviours grew out of the experience of early Erlang developers at Ericsson. Concurrency is hard. Fault tolerance is hard. The Erlang team put a lot of work into getting them right. Behaviours standardize their best practices and make them easy to use in our own applications.

OTP defines Behaviours for different types of specialized processes that we can use to build our own applications. We've already mentioned GenServer and Application, but there are others. There's one for finite state machines, one for creating and handling system events, and one for creating supervisors for fault tolerance. We can also define our own custom Behaviours to work in our own domains if we need to.

Each Behaviour is a module in OTP that contains the common code necessary for a process of that type. A GenServer, for instance, needs to be able to start new server processes, hold state, handle synchronous calls with a return value, as well as handle asynchronous casts without a return. The Behaviour module defines all that wiring and plumbing.

For real applications, the wiring and plumbing are not enough. We need to be able to customize a GenServer to handle the very specific requests our applications require.

Fortunately, there's an easy way to do this. We start by defining a normal module in our application. Then we inject the Behaviour module code into our new module. A Behaviour stores a list of callbacks—specific to it—that modules like ours need to implement in order to be an instance of that Behaviour. By writing those callbacks with code specific to our application, we customize our implementation of the Behaviour to work exactly the way we need it to.

That's the process we'll follow to build a GenServer for Islands.

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.

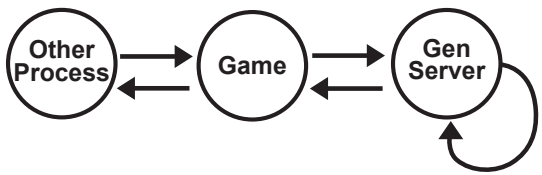
Let's try it out in the console.

```
~/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
20:49:47.773 [warn]  IslandsEngine.Game #PID<0.128.0> received unexpected message in handle_info/
```

That worked. At least it didn't throw an error. Here's why.

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 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.

The send/2 command worked for us because GenServer provides a default clause for handle_info/2 that matches any message, something like this.

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

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

```
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.

The warning we got is the compiler's way of telling us we need to implement our own handle_info/2 callback to override the default implementation.

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 Behavior that we don't need to send a message back to the caller, but we will return the state.

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.

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(game) do
  GenServer.call(game, :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. Then we'll use the map update syntax to set a new value for the state's :test key.

```
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(game, :demo)
:ok

iex> GenServer.call(game, :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.

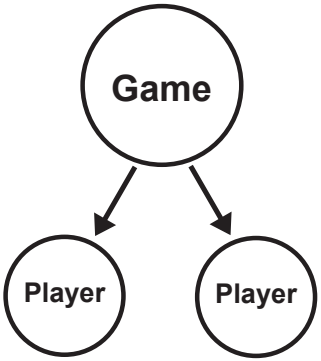
Initializing GenServer State

Until now, we've relied directly on GenServer's built-in functions for starting a new process. This works, but it doesn't help us define and initialize state very well. It would also be nice to have a really explicit way of starting a new game server by name, like we have for starting agents.

To do that, we've got some work to do. We'll need to define a struct to act as the GenServer state, add a public function that wraps GenServer.start_link/3, and define a callback function that start_link/3 triggers.

This ability to programmatically start new processes within the BEAM is key to the flexible, granular scaling we get with GenServer.

Let's start our implementation with the struct. We know that each game needs two players, like this.



Our struct will need two keys, :player1 and :player2. We've already defined Player agents in the last chapter. We can't start them at compile time, so we define them with the default value of :none. We'll see how to assign initialized Player agents as values for these keys very soon.

gen_server/lib/game.ex
```
defstruct player1: :none, player2: :none
```

We add IslandsEngine.Game and IslandsEngine.Player to our list of aliases.

gen_server/lib/game.ex
```
alias IslandsEngine.{Game, Player}
```

With the preliminaries out of the way, it's time to build in a way to start and initialize a new game.

We're about to add functionality to our GenServer, so we'll follow the pattern: write a client function that wraps a module function that triggers a callback.

We used start_link as the public function to start all our Agents, so lets do that again here for our client function. It needs to know who started the game so we can assign that name to the first player agent. We'll need to pass that in when we start new processes.

The GenServer module provides the start_link/3 module function to actually spawn the new process. It takes the name of the module to spawn, the argument to pass to the callback, and an optional list of options.

We'll use __MODULE__, a macro which returns the name of the current module, instead of hard-coding the module name.

While we're at it, we should make sure that the name isn't nil, so let's guard for that.

```
def start_link(name) when not is_nil name do
  GenServer.start_link(__MODULE__, name)
end
```

That covers the client and module functions. Now for the callback. GenServer.start_link/3 triggers GenServer.init/1.

The general form of init/1 is to pattern match on an argument, perform any necessary initializations, and return a tagged tuple of the form {:ok, initial_state}.

gen_server/lib/game.ex

```
def init(name) do
  {:ok, player1} = Player.start_link(name)
  {:ok, player2} = Player.start_link()
  {:ok, %Game{player1: player1, player2: player2}}
end
```

Notice we only set the name for the first player. One player starts the game. We'll show how we'll add the second player in a bit.

Let's see it in action.

In a new console session, first alias IslandsEngine.Game.

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

Then start a new game with the username "Frank", binding the variable game to the new game PID along the way.

```
iex> {:ok, game} = Game.start_link("Frank")
{:ok, #PID<0.99.0>}
```

Then use the Game.call_demo/1 to show the game state after startup.

```
iex> state = Game.call_demo(game)
%IslandsEngine.Game{player1: #PID<0.220.0>, player2: #PID<0.328.0>}
```

This is what we're looking for. We've got two player agents set in the game's state.

We can test that the first player has the name we sent in when we initialized the game by rendering that player agent as a string.

```
iex(5)> IO.puts IslandsEngine.Player.to_string(state.player1)
%Player{:name => Frank,
. . .
```

Which it did; that's exactly what we want.

Customizing GenServer Behavior

Spawning and initializing a new GenServer process is a great start, but the default callback implementations don't go very far. In order to build real applications, we need to change the way a GenServer behaves to fit each application's needs. We also need to provide a public interface for other processes to interact with.

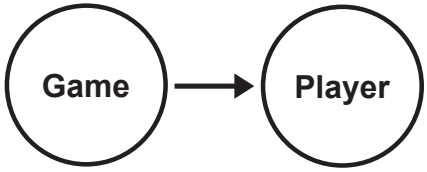
To do this, we're going to follow the pattern we've used so far—write a client function that wraps a GenServer function that triggers a callback. The callback is where we'll define the new behavior. The only way another process will be able to access any game server state or behavior is through the public interface we'll build with client functions. That ensures the encapsulation we're looking for.

In the last chapter we built a hierarchy of agents representing the data for a game. We noted that agents would need to communicate and cooperate in order to complete complex tasks. We'll see that in action again in this section in the form of chained function calls from agent to agent to get the job done.

Let's get to it.

Add a New Player

Each game will need to add a second player in order to begin play. This really amounts to assigning the second player agent a name. In this case, the chain of function calls is short, only two links. We have a function in the Player module to set a player agent's name property, but we'll need to access it through a public function in the Game module.



Let's call the client function `Game.add_player/2`. It should take the PID of the game and the name of the new player. We should also add a guard to make sure the name isn't nil.

gen_server/lib/game.ex

```
def add_player(pid, name) when name != nil do
  GenServer.call(pid, {:add_player, name})
end
```

We'll need a handle_call/3 clause that pattern matches for the {:add_player, name} tuple we passed into GenServer.call/3.

gen_server/lib/game.ex
```
def handle_call({:add_player, name}, _from, state) do
  Player.set_name(state.player2, name)
  {:reply, :ok, state}
end
```

All this needs to do is call the Player.set_name/2 function we've already defined.

Players are Agents

Notice we don't need to re-bind any values in the game state when we give the second player a name. Game struct keys map to Player PIDs, not their state. We can update a Player's state without changing its PID, so we don't have to re-bind any other data structures which hold them.

Let's see this in action. In a new console session, let's alias IslandsEngine.Game and start a new game server.

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

iex> {:ok, game} = Game.start_link("Frank")
{:ok, #PID<0.99.0>}
```

Now let's use our Game.call_demo/1 and Player.to_string/1 functions to take a peek at the state. Player1 has the name "Frank", but player2 doesn't yet have a name.

```
iex> state = Game.call_demo(game)
%IslandsEngine.Game{player1: #PID<0.220.0>, player2: #PID<0.328.0>}

iex> IO.puts IslandsEngine.Player.to_string(state.player1)
%Player{:name => Frank,
. . .

iex> IO.puts IslandsEngine.Player.to_string(state.player2)
%Player{:name => none,
. . .
```

Let's call the Game.add_player/2 function passing in the game PID and the name "Dweezil" and take another look.

```
iex> Game.add_player(game, "Dweezil")
:ok

iex> IO.puts IslandsEngine.Player.to_string(state.player1)
%Player{:name => Frank,
. . .
```

```
iex> IO.puts IslandsEngine.Player.to_string(state.player2)
%Player{:name => Dweezil,
. . .
```

Yay! It works.

Now that we can add a second player, it's time to have the game server handle adding coordinates to islands.

Set Island Coordinates

When players move their islands around on the board, the UI will send a list of coordinate tuples that the island occupies to the game server. Our GenServer will need to handle these requests and alter the game state accordingly.

Same pattern; we'll start with set_island_coordinates/4 for the client function. In addition to the player, island type and coordinate list, it will need to take the game server PID.

Ideally, the player and island type will both come in as atoms. That will make it easy to use Map functions to get the actual values we need. Let's put in guards for both of those.

gen_server/lib/game.ex

```
def set_island_coordinates(pid, player, island, coordinates)
  when is_atom player and is_atom island do

  GenServer.call(pid, {:set_island_coordinates, player, island, coordinates})
end
```

Since we're using the Coordinate module, let's add that to the aliases we're already using.

```
alias IslandsEngine.{Coordinate, Island, IslandSet}
```

The callback is where this will get interesting. We've got a big structure of agents to work with. We'll need to find the right island in the right island set belonging to the right player in order to update it. We could try to pull out the island we need in a single function with a series of Agent.get/2 calls, but that would be especially tedious.

Instead, let's create a chain of functions that lead from the game process to the island agent we care about, with each function doing just one part of the work before passing the task on to the next function. Here's the path we'll follow.

<imagedata fileref="images/gen_server/set_island_coords_path.pdf" />

The function calls will follow the lines of visibility and access. The game has access to the correct player. The player has access to the correct island set. That island set has access to the island we need to set coordinates for. At the end of the line, island agents already have a function for setting new coordinates.

gen_server/lib/game.ex
```
def handle_call({:set_island_coordinates, player, island, coordinates}, _from, state) do
  state
  |> Map.get(player)
  |> Player.set_island_coordinates(island, coordinates)
  {:reply, :ok, state}
end
```

We start by getting the right player agent from the game state and passing it along to the Player.set_island_coordinates/3 function. We haven't written that function yet, so let's do that next.

Player.set_island_coordinates/3 takes a player, and island key, and a list of coordinate atoms. We'll have a list of atoms, but the island will need a list of coordinate agents. Only the board knows how to convert coordinate atoms to agents.

Player agents have access to the player's board, so this seems like the right place to do the conversion. To make this work, we'll need two functions in the IslandsEngine.Player module to make this a little easier.

The first is Player.get_board/1 that will return a player's board agent.

gen_server/lib/player.ex
```
def get_board(player) do
  Agent.get(player, fn state -> state.board end)
end
```

The next is Player.get_island_set/1 that will return a player's island_set agent.

gen_server/lib/player.ex
```
def get_island_set(player) do
  Agent.get(player, fn state -> state.island_set end)
end
```

Now we're ready for Player.set_island_coordinates/3.

gen_server/lib/player.ex
```
def set_island_coordinates(player, island, coordinates) do
  board = Player.get_board(player)
  island_set = Player.get_island_set(player)
  new_coordinates = convert_coordinates(board, coordinates)
  IslandSet.set_island_coordinates(island_set, island, new_coordinates)
end
```

The last thing we do here is call IslandSet.set_island_coordinates/3. We'll get to that function shortly. For now, let's circle back to converting coordinates. Most of the time, we'll be passing this function an atom representation of a coordinate, so we'll need to use that atom to get the real coordinate agent to do any work.

gen_server/lib/player.ex

```
defp convert_coordinates(board, coordinates) do
  Enum.map(coordinates, fn coord -> convert_coordinate(board, coord) end)
end

defp convert_coordinate(board, coordinate) when is_atom coordinate do
  Board.get_coordinate(board, coordinate)
end
defp convert_coordinate(_board, coordinate) when is_pid coordinate do
  coordinate
end
```

Our strategy will be to enumerate over the list of coordinates and convert each one individually, returning a list of coordinate agents. Enum.map/2 is perfect for this. We'll use a helper function to do the actual conversion.

In the common case, where the coordinate is an atom, we use the Board.get_coordinate/2 function we already have to get the coordinate agent and return it. In the unlikely event that the coordinate already is a PID, meaning that it's an agent, we just return it.

Now that we have that set up, let's move on to IslandSet.set_island_coordinates/3.

When we set coordinates in an island we replace the existing list of coordinates with a new list. Coordinates keep track of whether they are in an island or not. This means that when we set new coordinates, we displace the old ones. Those old ones will still believe that they are in an island unless we do something about it.

Here's what we'll do. Before we set new coordinates, we'll get the list of current ones. After we replace the current coordinates, we'll mark them all as no longer in an island. Then we'll mark all the new ones as in an island.

gen_server/lib/island_set.ex

```
def set_island_coordinates(island_set, island_key, new_coordinates) do
  island = Agent.get(island_set, fn state -> Map.get(state, island_key) end)
  original_coordinates = Agent.get(island, fn state -> state end)
  Island.replace_coordinates(island, new_coordinates)
  Coordinate.set_all_in_island(original_coordinates, :none)
  Coordinate.set_all_in_island(new_coordinates, island_key)
end
```

There's one new function we can define to make this easier, Coordinate.set_all_in_island/2. This enumerates over a list of coordinates and sets the :in_island? value to the one we supply.

gen_server/lib/coordinate.ex
```
def set_all_in_island(coordinates, value)
  when is_list coordinates and is_atom value do
  Enum.each(coordinates, fn coord -> set_in_island(coord, value) end)
end
```

Say, don't I recognize you from somewhere?

Students of Object Oriented Design patterns will recognize what we've just done as upholding The Law of Demeter—relying on a collaborator's public interface, not its implementation. This keeps us out of trouble if the implementation changes. It works in Elixir because if you squint just right, Elixir processes look a lot like Alan Kay's original idea for objects.

Now let's see how it works. Open up a new console session, alias the IslandsEngine.Game module, and start up a new game.

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

iex> {:ok, game} = Game.start_link("Frank")
{:ok, #PID<0.111.0>}
```

We'll set a list of a single coordinate to player1's :dot island. The client function expects the player and island arguments to be atoms, so let's make sure to provide them.

```
iex> Game.set_island_coordinates(game, :player1, :dot, [:a1])
:ok
```

Now we can look at player1's state to see what happened. We'll cut out all the extra info and focus on the :dot island and the :a1 coordinate.

```
iex> state = Game.call_demo(game)
%IslandsEngine.Game{player1: #PID<0.231.0>, player2: #PID<0.339.0>}

iex> IO.puts IslandsEngine.Player.to_string(state.player1)
. . .
dot => [(in_island:dot, guessed:false)]
. . .
:board => %{a1 => (in_island:dot, guessed:false),
. . .
```

That's great news right away. The :dot island has a single coordinate agent, and the :a1 coordinate says it's the one. The atom to coordinate conversion worked, and we've correctly set the coordinate list.

Now let's set another list of a single coordinate on the :dot island.

```
iex> Game.set_island_coordinates(game, :player1, :dot, [:b3])
:ok
```

Ok, let's see what player1's state looks like now.

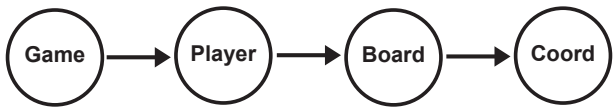
```
iex(8)> IO.puts IslandsEngine.Player.to_string(state.player1)
. . .
dot => [(in_island:dot, guessed:false)]
. . .
:board => %{a1 => (in_island:none, guessed:false),
. . .
b3 => (in_island:dot, guessed:false),
. . .
```

That's perfect. The :dot island has a single coordinate, and the :b3 coordinate says it's the one. The :a1 coordinate is no longer in any island.

That all works as it should. The next thing we need to tackle is guessing a coordinate.

Guess a Coordinate

Guessing coordinates is the most important action in the game of Islands. It seems simple, but there's a lot going on. We need to find a path from the player doing the guessing to a specific coordinate on their opponent's board. We've got to mark that coordinate as guessed and return whether it's a hit or a miss. We'll definitely use a chain of functions for this.



As always, we'll need a client function. Let's call it guess_coordinate/3. A guess requires a player and a coordinate. As we've seen, both players and coordinates should be atoms. We'll add guards for those constraints.

gen_server/lib/game.ex

```
def guess_coordinate(pid, player, coordinate)
  when is_atom player and is_atom coordinate do
```

```
  GenServer.call(pid, {:guess, player, coordinate})
end
```

When a player guesses a coordinate, they are really guessing a coordinate on their opponent's board. We'll talk about the opponent/2 function that finds a player's opponent in a minute.

```
def handle_call({:guess, player, coordinate}, _from, state) do
  opponent = opponent(state, player)
  opponent_board = Player.get_board(opponent)
  response = Player.guess_coordinate(opponent_board, coordinate)
  {:reply, response, state}
end
```

Once we have the opponent, we can pass their board and the coordinate key into Player.guess_coordinate/2, a function we have yet to write.

Let's go back to Game.get_opponent/2. We pass it the game state and a player. A pattern match and two function clauses do all the work.

gen_server/lib/game.ex

```
defp opponent(state, :player1) do
  state.player2
end
defp opponent(state, _player2) do
  state.player1
end
```

The last function in this chain is Player.guess_coordinate/2, which is where the most work gets done. It determines whether the guess was a hit or a miss.

gen_server/lib/player.ex

```
def guess_coordinate(opponent_board, coordinate) do
  Board.guess_coordinate(opponent_board, coordinate)
  case Board.coordinate_hit?(opponent_board, coordinate) do
    true -> :hit
    false -> :miss
  end
end
```

Time to see how this works! Let's begin with a new console session.

```
iex> alias IslandsEngine.{Game, Player}
[IslandsEngine.Game, IslandsEngine.Player]

iex> {:ok, game} = Game.start_link("Frank")
{:ok, #PID<0.110.0>}
```

Now we're ready for a player to guess a coordinate. We'll have player2 guess the A1 coordinate this time.

```
iex> Game.guess_coordinate(game, :player2, :a1)
```

```
:miss
```

It works! It's a miss because we haven't added :a1 to any islands yet.

Let's verify that :a1 has the value true for :guessed?. Start by aliasing IslandsEngine.Board. Then get player1's board agent. Remember that player2 did the guessing, so the coordinate came from the opposite board.

```
iex> state = Game.call_demo(game)
%IslandsEngine.Game{player1: #PID<0.221.0>, player2: #PID<0.329.0>}

iex> player1 = state.player1
#PID<0.221.0>
```

Now let's take a look at player1's state.

```
iex> IO.puts Player.to_string(player1)
. . .
:board => %{a1 => (in_island:none, guessed:true),
. . .
```

Success! player2 did guess player1's :a1 coordinate.

Pipelines to Handle Complexity

Often we have an operation that's really complex. We might be tempted to handle it with a single function, but that's usually a mistake. Better to break the steps down into their own functions with meaningful names, and then compose the whole operation together again using the smaller functions. By breaking the big function up, we gain clarity via the named functions that describe each part of the operation. The smaller functions are easier to understand and modify, and we can keep their idea in our head.

Guessing a coordinate works well, but there is room for a little improvement. Let's imagine we're playing the game. When a guess is a hit, there are two questions that immediately follow, "Did that hit forest an island?" and "If one was forested, did it win the game?"

We could expect the UI to answer these questions by making extra calls to the server. That would add latency, and it's not necessary. We have all the information we need to provide a complete response up front.

We can standardize the response to a guess as a three element tuple. The first element will be either :hit or :miss. The next element would be either an atom representing the forested island name or :none. The third element will be either :win or :no_win.

Player.guess_coordinate/2 already handles the first element, but we'll need new functions for the second and third elements.

We should only do the work we need to and no more. If the guess was a miss, we shouldn't check to see if the island was forested because it couldn't possibly be. If a guess was a hit but the island was not forested, we don't need to check for a win.

There's a fair bit of work involved with formulating the full return tuple. Up to now, we've used function composition to simplify complex tasks. This time, though, we'll add in an Elixir pipeline and data transformation to shape the return we want, one step at a time.

Check for a Forested Island

Our task for this section is clear. We need to figure out if a guess results in a forested island and add that result to the return tuple we're building up. We'll need a new Game.forest_check/3 private function to help us.

We'll rely on a pipeline to build up the return tuple, but we'll still need the other tools we've used so far—composition, pattern matching, and multi-clause functions.

Let's start by piping the response from Player.guess_coordinate/2 into Game.forest_check/3. Game.forest_check/3 will also need the opponent and the guessed coordinate to get the key for the island it is in.

```
def handle_call({:guess, player, coordinate}, _from, state) do
  opponent = opponent(state, player)
  opponent_board = Player.get_board(opponent)
  response = Player.guess_coordinate(opponent_board, coordinate)
  |> forest_check(opponent, coordinate)
end
```

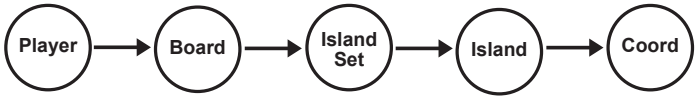
We haven't written Game.forest_check/3 yet, so let's do that now.

gen_server/lib/game.ex

```
defp forest_check(:miss, _opponent, _coordinate) do
  {:miss, :none}
end
defp forest_check(:hit, opponent, coordinate) do
  island_key = Player.forested_island(opponent, coordinate)
  {:hit, island_key}
end
```

The first clause is easy. If the guess was a miss, it could not have forested an island, so we know the return will be {:miss, :none}.

The second clause does more work. It needs to find the island that contains the guessed coordinate, and ask that island if it is forested. We'll write a new chain of functions to do this, traversing the Player, Board, IslandSet, and Island modules.



The Player module knows how to find the board and island set, so let's start there with a new forested_island/2 function.

gen_server/lib/player.ex

```
def forested_island(opponent, coordinate) do
  board = Player.get_board(opponent)
  island_key = Board.coordinate_island(board, coordinate)
  island_set = Player.get_island_set(opponent)

  case IslandSet.forested?(island_set, island_key) do
    true -> island_key
    false -> :none
  end
end
```

The first thing that Player.forested_island/2 needs to do is get the name of the island from the coordinate. Then it can ask the island set if that island is forested. If it is forested, we return the island key, otherwise we return :none.

gen_server/lib/island_set.ex

```
def forested?(_island_set, :none) do
  false
end
def forested?(island_set, island_key) do
  island_set
  |> Agent.get(fn state -> Map.get(state, island_key) end)
  |> Island.forested?
end
```

Game.forest_check/2 takes that island key and returns a two element tuple to return. That will look something like this {:hit, :dot} or this {:miss, :none}.

With that, we're ready for the final function in this pipeline.

Check for a Win

We're at the third and final stage of this pipeline, determining whether a guess results in a win or not. This is where we'll complete the return tuple, and we'll use all the techniques we've learned so far to do it.

Let's start by piping the return from Game.forest_check/3 into a new private function, Game.win_check/3.

gen_server/lib/game.ex

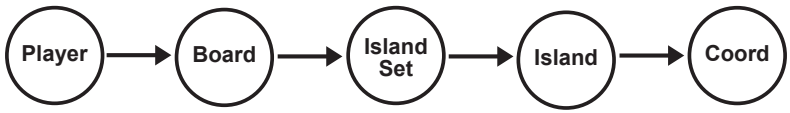
```
def handle_call({:guess, player, coordinate}, _from, state) do
  opponent = opponent(state, player)
  Player.guess_coordinate(opponent.board, coordinate)
  |> forest_check(opponent, coordinate)
  |> win_check(opponent, state)
end
```

With some clever pattern matching and multiple clauses, Game.win_check/3 can avoid some unnecessary work.

gen_server/lib/game.ex

```
defp win_check({hit_or_miss, :none}, _opponent, state) do
  {:reply, {hit_or_miss, :none, :no_win}, state}
end
defp win_check({:hit, island_key}, opponent, state) do
  win_status =
  case Player.win?(opponent) do
    true -> :win
    false -> :no_win
  end
  {:reply, {:hit, island_key, win_status}, state}
end
```

The first clause practically writes itself. No matter whether a guess is a hit or a miss, if it doesn't forest an island, the player can't win.



The remaining clause handles the case of both a hit and a forested island. Sometimes this will be a win and sometimes it won't. To determine which it is, we'll need a new function in the Player module, Player.win?/1.

gen_server/lib/player.ex
```
def win?(opponent) do
  opponent
  |> Player.get_island_set()
  |> IslandSet.all_forested?()
end
```

Winning means foresting all your opponent's islands. Island sets contain all a player's islands, so we'll need a new IslandSet.all_forested?/1 to enumerate over all of them and tell us if they're all forested or not.

gen_server/lib/island_set.ex
```
def all_forested?(island_set) do
  islands = Agent.get(island_set, &(&1))
  Enum.all?(keys(), fn key -> Island.forested?(Map.get(islands, key)) end)
end
```

The Enum.all?/2 Function

Enum.all?/2 takes any data structure that implements the Enumerable protocol, like a list, and an anonymous function that returns a boolean to apply to each element of the Enumerable. If the anonymous function replies true for all the elements, Enum.all?/2 replies true. Otherwise, it replies false.

It's been a while since we've checked anything in the console. Let's fix that by verifying all the work we've just done. We'll set up player1 with some islands and have player2 guess to show all the possible combinations.

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

iex> {:ok, game} = Game.start_link("Sarah")
{:ok, #PID<0.92.0>}

iex> Game.add_player(game, "Ben")
:ok

iex> Game.set_island_coordinates(game, :player1, :atoll, [:a1, :a2])
:ok

iex> Game.set_island_coordinates(game, :player1, :dot, [:c1])
```

Now for some guesses. Let's see what a miss looks like.

```
iex> Game.guess_coordinate(game, :player2, :d5)
{:miss, :none, :no_win}
```

That's great. Now let's see a hit without a forested island.

```
iex> Game.guess_coordinate(game, :player2, :a1)
{:hit, :none, :no_win}
```

Beautiful. This time let's try a hit that forests player1's atoll, but doesn't win.

```
iex> Game.guess_coordinate(game, :player2, :a2)
{:hit, :atoll, :no_win}
```

That's exactly what we want. Now let's try a hit that forests player1's remaining island and wins the game.

```
iex> Game.guess_coordinate(game, :player2, :c1)
{:hit, :dot, :win}
```

That's perfect. Now with each guess we'll be able to return all the information the UI will need to communicate to the players.

Naming GenServer Processes

This whole chapter we've been starting new GenServer processes and binding their PIDs to variables. Whenever we've needed to call a public function on a process, we've passed that variable in as the first argument. That clearly works, but it leaves us with a problem.

In a full application, we'd need to keep track of every variable for every process we started. We'd need to always keep them in scope, and we'd need to clear individual variables out when their process stopped. If that sounds like a serious hassle, it is.

It would be great if we could just name each process as we start it, and pass that name in whenever we wanted to call a function. It would be even better if we didn't have to remember the name, but were able to reconstruct it on the fly when we needed to. If wishes came true, these names would clear themselves out when their process stopped.

Wishes can come true. Process registration will do all of this for us.

There are several ways to register GenServer processes by name. Let's explore them and see which best fits our needs.

The first thing we can do is simply specify a name as an atom as the third argument to GenServer.start_link/3. Let's open up a new console to test that out.

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

iex> GenServer.start_link(Game, {:ok, "Frank"}, name: :islands_game)
{:ok, #PID<0.90.0>}
```

We're fine using the raw GenServer.start_link/3 function here. We need to specify the module name instead of using __MODULE__ because this function call isn't originating inside the Game module the way it is in IslandsEngine.Game.start_link/1.

The part to watch is the keyword list we specified as the third argument, name: :islands_game. This clearly works. We get {:ok, #PID<0.90.0>} as the return value.

Now we can use the atom :islands_game instead of a PID whenever we call client functions in the Game module.

```
iex> Game.call_demo(:islands_game)
%IslandsEngine.Game{player1: #PID<0.221.0>, player2: #PID<0.329.0>}
```

The atom has a direct one to one mapping to the PID of a single GenServer. If we try to start another one with the same atom for the name, we get an error saying that the server is already running.

```
iex> GenServer.start_link(Game, {:ok, "Frank"}, name: :islands_game)
{:error, {:already_started, #PID<0.90.0>}}
```

Using Process Registration to Our Advantage

This property of naming GenServers can be useful. If we ever need to enforce that there be only a single instance of a given GenServer, we can name it with a hard-coded atom. The Erlang virtual machine will not allow more than one GenServer of that type to start up.

This type of process registration is called a local name. It is only visible on the same node on which the process is spawned, and the name must be an atom.

This leads to a problem for our use case. In Islands, we'll be spinning up a new game process for each pair of players. Elixir doesn't garbage collect atoms so the list of atoms will grow as we spawn more games. The BEAM enforces a hard limit of about a million atoms. If we reach that limit, the whole node will crash, no exceptions. Those are the kinds of things that trigger system alerts in the middle of the night.

We need another way. Fortunately, we can register a process name as a string with Erlang's global name service. This hardly requires any change at all. We just specify the value of the name as a tagged tuple.

The one bit of data we have whenever we start a new game is the first player's name. That's a value we'll have around as long as the game exists, and it's perfect to construct a name with.

```
iex> GenServer.start_link(Game, {:ok, "Frank"}, name: {:global, "game:Frank"})
{:ok, #PID<0.306.0>}
```

The global name registry works across all connected nodes in the system. If we add more nodes, they will automatically know about—and be able to use—the globally registered processes.

The global name registry will also take care of re-mapping PIDs when supervisors restart their processes as well as removing entries for terminated processes.

The :via Option

If we want, we can define our own module to register process names in any way that fits our purpose. Any module that defines register_name/2, unregister_name/1, whereis_name/1, and send/2 will do. With a module like that in place, we can register our process name with {:via, module_name, term_for_name}. :global is the name of one such module that happens to be built into Erlang.

The global registry won't let us register the same string for a new process. If we try, we get the same error as when we tried to register the same local name twice.

```
iex> GenServer.start_link(Game, {:ok, "Frank"}, name: {:global, "game:Frank"})
{:error, {:already_started, #PID<0.306.0>}}
```

We're working with a string, so we can use variable interpolation.

```
iex> name = "Dweezil"
"Dweezil"

iex> GenServer.start_link(Game, name, name: {:global, "game:#{name}"})
{:ok, #PID<0.113.0>}
```

We can then use the full tagged tuple, including the interpolated string, as a stand-in for the PID whenever we call any IslandsEngine.Game functions.

```
iex> Game.call_demo({:global, "game:Dweezil"})
%IslandsEngine.Game{player1: #PID<0.221.0>, player2: #PID<0.329.0>}
```

This leads us to a new implementation of IslandsEngine.Game.start_link/1 which will give us a unique global server name based on the name we assign to player1. This also means that a player can only start a single game at a time, but that's fine for our purposes.

gen_server/lib/game.ex

```
def start_link(name) when is_binary(name) and byte_size(name) > 0 do
  GenServer.start_link(__MODULE__, name, name: {:global, "game:#{name}"})
end
```

This is a simple solution which might not be appropriate for all applications. If we needed to, we could generate a unique string using any method we deemed appropriate and secure.

Stopping GenServer Processes

There's one last thing we need to take care of. We should be able to gracefully stop the GenServer. This will not only free up system resources, but also remove the server's name from the global registry, allowing the initiating player to start another game.

Back to the OTP pattern, we'll start with a client function.

gen_server/lib/game.ex

```
def stop(pid) do
  GenServer.cast(pid, :stop)
end
```

The callback implementation is really simple.

gen_server/lib/game.ex
```
def handle_cast(:stop, state) do
  {:stop, :normal, state}
end
```

GenServer sees the tag :stop and takes care of all the work of shutting the process down and cleaning up after it.

Let's see it in action in a new console session.

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

iex> Game.start_link("Frank")
{:ok, #PID<0.104.0>}
```

Let's make sure we can't start another game with the same name.

```
iex> Game.start_link("Frank")
{:error, {:already_started, #PID<0.1.0>}}
```

That returns an error as we expected.

Now let's just show that the server is working by calling Game.call_demo/1. Then we'll stop the server and try Game.call_demo/1 again.

```
iex> Game.call_demo({:global, "game:Frank"})
%IslandsEngine.Game{player1: #PID<0.221.0>, player2: #PID<0.329.0>}

iex> Game.stop({:global, "game:Frank"})
:ok

iex> Game.call_demo({:global, "game:Frank"})
** (exit) exited in: GenServer.call({:global, "game:Frank"}, :demo, 5000)
    ** (EXIT) no process: the process is not alive or there's no process
    currently associated with the given name,
    possibly because its application isn't started
    (elixir) lib/gen_server.ex:729: GenServer.call/3
```

That's exactly the error we want. By calling Game.stop/1, we shut the server down and made it unavailable for the next call.

Now that the server is gone, the name should no longer be in the global registry. We should be able to start a new game server with the same name.

```
iex> Game.start_link("Frank")
{:ok, #PID<0.321.0>}
```

We can! That's exactly what we want.

Wrapping Up

We've come a long way in this chapter.

We've learned about OTP Behaviours. That knowledge will stand us in good stead for the next few chapters as well as all of part two. More importantly, it'll come in handy whenever we're building applications in Elixir.

We've built a custom GenServer. We wrote functions to start and stop it automatically. We drilled the pattern of client function wrapping a GenServer function which triggers a callback into our heads. And we saw how to register names to individual GenServer processes so that they are addressable from anywhere.

The game itself has come a long way. We used function chains and composition to distribute complex behavior across the structure of agents we built up in the last chapter. It's really starting to look like a game!

We're not done yet, though. A game will go through a number of states between startup and one player winning. We'll want to be able to track these states and enforce different rules accordingly. For example, players can move their islands around on the board until the players both say they are set. After that, neither player can move any of their islands.

We'll tackle these problems with a finite state machine. That will be our task in the next chapter.

What we'll do in this chapter

- *implement a :gen_statem Behaviour*
- *use :gen_statem to model the rules of Islands*
- *cleanly integrate the :gen_statem module into our GenServer*

CHAPTER 4

Manage State with gen_statem

Handling state is a hot topic in web development these days. We're seeing changing ideas and practices in both the front and back end worlds. It's about time we talk about state more directly.

The BEAM's concurrency and fault tolerance bring truly stateful Web applications within reach. But stateful applications bring their own challenges. Managing state over time requires great care and coordination. Keeping code clean in the process provides an extra level of difficulty.

We'll meet these challenges with an OTP Behaviour called :gen_statem. We'll see how to use :gen_statem to make decisions and enforce rules in an application. :gen_statem will help us coordinate events and transitions as well. Most importantly, we'll keep our code clean by separating state management from business logic.

Before we move on to where we're going, let's see where we've been.

A Bit of History

Early client-server applications were stateful. Clients connected to the server and stayed connected while they passed messages back and forth. Think mainframe applications with dedicated terminals as clients.

That worked well, but it meant that the number of possible clients was limited by system resources like memory, cpu, and the number of concurrent processes the system could support.

When Tim Berners-Lee invented the web, he found a way around those limitations. He made HTTP a stateless protocol. When a client makes an HTTP request to a server, it must supply all the data the server will need to fulfill that request. Once the server sends its response, it forgets everything it just knew about both the request and the client.

This request-response cycle has been critical to the success and scaling of the web. It requires fewer system resources to handle vastly more requests because the server doesn't need to keep track of anything once it sends a response. This allows applications to use less expensive shared pools for resources like database or mainframe connections instead of more expensive dedicated resources for each client. Applications can manage other resources like threads and memory the same way.

HTTP shed resource costs, but it picked up others along the way. It's impractical to pass all the state a complex application needs to do its work. Instead, servers store that state in a database, and clients pass along only enough information for the server to fetch that data to fulfill the request. If the request involves any change in state, the server needs to write those changes back to the database. These trips to and from the database add latency. Modeling a domain for a database adds unnecessary complexity.

As developers, we pick up the tab for these added costs in terms of extra code to write and maintain as well as extra cognitive load when reasoning about our applications.

Change is Afoot

As applications grow and traffic increases, these costs begin to really add up. At serious scale, they can become prohibitive, so people are looking for ways to get around them.

We're at the beginning of a sea change in web development. We're seeing the return of stateful servers with persistent client connections. Modern hardware provides abundant system resources. Elixir provides more than enough power and concurrency to handle application state and persistent connections at scale. Phoenix Channels make writing those persistent connections a breeze.

With stateful applications, we no longer have the luxury of clean state with every new request. We have to manage state over time, and make sure it remains consistent. We need to understand the stages an application can go though, and handle the transitions between them. We need to ensure that events in the system are consistent with the stage the application is in.

The JavaScript World

JavaScript developers have already walked this path from a mostly stateless to stateful environment. With the rise of AJAX requests a number of years ago, frontend web applications could fetch data outside of the normal request-

response cycle. They could update the DOM without a full page reload that would wipe the state clean.

The rise of AJAX opened up incredible possibilities in user interactions. Web applications became as complex and interesting as desktop apps—map applications, email, and office suites. With the vastly increased time between page reloads, the browser suddenly became a stateful environment, and developers quickly found out that they needed strategies to manage state.

The JavaScript world is still grappling with this shift. The community is continually inventing new solutions to ease the difficulty of handling state—frameworks, data binding libraries, promise libraries, generators, and more. The sheer number of solutions out there and the speed with which they hit the ecosystem creates a very real sense of fatigue.

A Different Path

You might think that we could make decisions about application stages, stage transitions, and events with conditional logic. You would be right, but the costs would be high. The number of nested "if" statements necessary to do the job would lead to a snarl of code paths. Real readability and maintainability problems would be our reward.

There's a better way. Finite state machines are tailor made for managing complex state over time, and OTP gives us not one, but two really good ones. There's an older one called `:gen_fsm` which is still included with OTP. There's also a recent addition called `:gen_statem` that the OTP team suggests using for new projects going forward. That's what we'll use.

We're going to implement our own instance of `:gen_statem` to handle all the stages that Islands will go through in the course of a full game. It's going to make decisions for the application about which actions to allow and which to deny at a given stage. It will manage transitions from one stage to the next, and it will help the game enforce the rules.

By implementing our own `:gen_statem`, we'll keep the logic for managing state separate from the rest of the game logic. This separation of concerns will keep our code clean, readable, and maintainable.

Before we go any further, let's get a better understanding of state machines in general and take a look at `:gen_statem` in particular. Along the way, we'll see an example of a problem that's ideal for a state machine to solve.

State Machines and :gen_statem

State machines are fundamental to software development. Any time we need to model a complex process that proceeds through a number of states, especially ones that might loop back to earlier states, we should think about reaching for a finite state machine.

state vs. state

We need to resolve an ambiguity with the word "state." In previous chapters, when we said "state," we meant data held in an Elixir process. In a state machine, a "state" is the name of a stage in an application. In this chapter, we'll use the term "state data" to refer to data held by an Elixir process, and we'll use "state" to mean the name of a stage in an application.

Let's walk through an example of a problem that's perfect for a state machine.

Imagine you're working at an e-commerce site with its own warehouse. As part of your job, you need to model and control the lifecycle of a product from the first time a buyer sees it at a trade show until it's stocked in the warehouse and available for sale.

There are a surprising number of states a product can go through. If a buyer sees a product they like and adds it to the system, that first state could be called `scouted`. After that, a buyer might order a sample. We could call that state `sample_ordered`. Once the sample arrives the buyer might accept it and order a larger number of them. We might call that state `inventory_ordered`. The buyer might also reject the sample and have the state go back to `scouted`.

There are rules here, and a sense of progression. Buyers can't order a sample unless they've scouted the product. They can't order inventory until they've ordered a sample, and so on.

This process might continue until the buyer accepts the production run, the warehouse completes the intake of the merchandise, and the warehouse has stocked the product in preparation for fulfillment. It might also stop at any point and go back to a previous state if something goes wrong.

This scenario demonstrates the transitions between states, but it glosses over a really important component. There are events in the system that trigger the state transitions. Buyers add products to the system. That's an event. Buyers order samples. Shipments arrive. Buyers accept samples, and so on. All events.

These are the keys to understanding state machines. Events trigger state transitions. The state machine may progress to a new state or regress to a previous one depending on the event and the current state.

In the Elixir community, we're lucky. OTP gives us a state machine Behaviour for free. It's called :gen_statem, and it works a lot like GenServer.

The first thing we can see is that :gen_statem has an unusual name. It's not an Elixir module. It comes directly from Erlang, and we address Erlang modules with their atom representations in Elixir. Even though it isn't an Elixir module, we can use it directly in any Elixir application.

We got to know GenServer pretty well in the last chapter. Both GenServer and :gen_statem are Behaviours, the design patterns/modules at the heart of OTP. They share some common characteristics.

- Both require us to write client functions which serve as the module's public interface.
- Both use module functions within those client function.
- Both require us to implement clauses of specific callbacks.
- Both trigger those callbacks in response to specific module functions.
- Both can handle either synchronous or asynchronous calls, depending on whether the caller needs a response.

GenServer and :gen_statem have differences as well.

Elixir wraps the raw Erlang :gen_server module in an Elixir module called GenServer. GenServer takes care of the boilerplate code needed to properly implement a :gen_server. Elixir doesn't wrap :gen_statem, so we'll have to do that work ourselves.

In :gen_statem, client functions represent events, things like adding a player or guessing a coordinate. As with GenServer, client functions wrap :gen_statem module functions, and those module functions trigger specific callbacks, passing along the event.

In :gen_statem, there are two types of callbacks which are mutually exclusive. We need to choose one type or the other when we set up our :gen_statem module.

The first is the event style, with a single callback function called handle_event/4. Each clause of handle_event/4 pattern matches on arguments that represent the event and the state name in order to map events to states.

Our other choice is the state function style, with multiple callback functions, each named after a state in the state machine. All of these will need to pattern match on an argument that represents the event. Since we name the function after the state, the event is all we need to map the event to the state.

We'll use the latter, state function style, because of it's clarity. Naming callbacks after their states makes it really easy to see which code will be executed for each state.

Both callback styles define the states. Part of their return value is always the next state to transition to. If a callback returns a next state that had not previously existed, we suddenly have a new state. There is no configuration step to name all the possible states up front.

The callbacks we write will decide whether an event is OK in the current state. They will also determine which state to transition to. The next state may be the current state if no transition is necessary.

:gen_statem gives us a lot of control over how we work with events and when transitions happen. We will be using synchronous replies throughout this chapter, but it's worth taking a look at the documentation to see what's possible.[1]

This may seem a little abstract at the moment. Hang in there. It'll become clear as we go along. We'll see these interactions between events, callbacks, and state a number of times in this chapter.

We're ready to begin implementing our own state machine using :gen_statem. We'll define a new module for it, get that module code running in it's own process, and then have our game server check in with that process before each action to make sure it's permissible.

Let's get started!

Getting Started with :gen_statem

Implementing a state machine with :gen_statem looks a lot like implementing a GenServer. We'll define our own module in our own application. Then we'll inject the code from the OTP :gen_statem module into ours. That will make our module behave like an instance of :gen_statem.

We'll customize our state machine with public functions and callbacks to make it behave the way we need it to. Along the way, we'll implement generic

1. http://erlang.org/doc/design_principles/statem.html

versions of some callbacks that OTP requires of an instance of :gen_statem, but for which we won't need any customized behavior.

Like Agents and GenServers, instances of :gen_statem run in separate processes. We'll need a programatic way of starting these processes, and that's what we'll focus on in this section.

Small, isolated processes provide much of the power of Erlang and Elixir. Most often, that means concurrency and fault tolerance. Here, what it means is separation of concerns. We'll be able to easily separate the game logic code from state handling code. This is going to keep our game clean, easy to read, and maintainable.

Fortunately, we won't have to reinvent the wheel here. Since GenServer and :gen_statem are OTP Behaviours, they work in very similar ways. Much of this section will seem familiar from our discussion of starting GenServers.

Let's start implementing our :gen_statem Behaviour. To begin, we'll need a module declaration in a new file at lib/rules.ex.

```
defmodule IslandsEngine.Rules do
  @behaviour :gen_statem

  alias IslandsEngine.Rules
end
```

After the module definition, we declare that this module will be a :gen_statem. Each Behaviour knows which callback functions are necessary to complete the Behaviour. Our implementation will need to include at least one clause of all the callback functions :gen_statem expects. The compiler will let us know which ones we're missing as we go along.

@behaviour :gen_statem

@behaviour looks like a normal module attribute, but it's really a reserved attribute to annotate the module and tell the compiler which Behaviour it will implement.

:gen_statem modules define server processes. We need to start these processes in order for them to work. Just as we did with Agents and GenServer, we'll need a start_link/0 client function wrapping :gen_statem.start_link/3.

gen_statem/lib/rules.ex

```
def start_link do
  :gen_statem.start_link(__MODULE__, :initialized, [])
end
```

As with GenServers, the :gen_statem.start_link/3 function triggers a corresponding init/1 callback.

```
def init(:ok) do
 {:ok, :initialized, []}
end
```

The only argument init/1 takes is the :ok atom we passed into start_link/3. We'll return :initialized for the initial state as well as the initial state data which we'll define as an empty list for now.

Let's check to make sure we can start a new process.

```
$ iex -S mix
Erlang/OTP 19 [erts-8.2] [source] [64-bit] [smp:8:8] [async-threads:10]
              [hipe] [kernel-poll:false] [dtrace]

Compiling 1 file (.ex)
warning: undefined behaviour function callback_mode/0 (for behaviour :gen_statem)
  lib/rules.ex:1

warning: undefined behaviour function code_change/4 (for behaviour :gen_statem)
  lib/rules.ex:1

warning: undefined behaviour function terminate/3 (for behaviour :gen_statem)
  lib/rules.ex:1

Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)

iex> alias IslandsEngine.Rules
IslandsEngine.Rules

iex> {:ok, pid} = Rules.start_link()
** (EXIT from #PID<0.121.0>) :undef

Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)
iex>
  09:41:52.434 [error] ** State machine #PID<0.124.0> terminating
  ** Last event = {:internal, :init_state}
  ** When server state  = {:initialized,
   %IslandsEngine.Rules{player1: :islands_not_set, player2: :islands_not_set}}
  ** Reason for termination = :error:{:"function not exported",
     {IslandsEngine.Rules, :terminate, 3}}
  ** Callback mode = :undefined
  ** Stacktrace =
  **  [{:gen_statem, :terminate, 6, [file: 'gen_statem.erl', line: 1492]},
  {:proc_lib, :init_p_do_apply, 3, [file: 'proc_lib.erl', line: 247]}]
```

Clearly, that didn't work at all. The direct cause of this crash is that we haven't defined one of the callback functions that the :gen_statem Behaviour requires, terminate/3.

```
Reason for termination = :error:{:"function not exported",
   {IslandsEngine.Rules, :terminate, 3}}
```

But the compiler warned us about two other missing callbacks as well, callback_mode/0 and code_change/4.

```
warning: undefined behaviour function callback_mode/0 (for behaviour :gen_statem)
  lib/rules.ex:1

warning: undefined behaviour function code_change/4 (for behaviour :gen_statem)
  lib/rules.ex:1
```

When we worked with GenServer, we said that it's the Elixir module that wraps :gen_server, the Erlang module. GenServer takes care of default implementations of the Behaviour callbacks for us. We don't have that luxury with :gen_statem, so we'll need to write these ourselves. Luckily, we can write simple, one-line functions for each of these three callbacks to satisfy the compiler.

Let's take them in the order of the compiler warnings.

The first one we need to tackle is callback_mode/0. This tells :gen_statem which type of callbacks we're choosing. We decided on using the state functions, so let's specify that here.

gen_statem/lib/rules.ex

```
def callback_mode(), do: :state_functions

def code_change(_vsn, state_name, state_data, _extra) do
  {:ok, state_name, state_data}
end

def terminate(_reason, _state, _data), do: :nothing

def initialized({:call, from}, :add_player, state_data) do
  {:next_state, :players_set, state_data, {:reply, from, :ok}}
end

def initialized({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :initialized}}
end

def initialized({:call, from}, _, _state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end

def players_set({:call, from}, {:move_island, player}, state_data) do
  case Map.get(state_data, player) do
    :islands_not_set ->
      {:keep_state_and_data, {:reply, from, :ok}}
    :islands_set ->
      {:keep_state_and_data, {:reply, from, :error}}
  end
end

def players_set({:call, from}, {:set_islands, player}, state_data) do
  state_data = Map.put(state_data, player, :islands_set)
  set_islands_reply(from, state_data, state_data.player1, state_data.player2)
```

```
end

def players_set({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :players_set}}
end

def players_set({:call, from}, _, state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end

def player1_turn({:call, from}, {:guess_coordinate, :player1}, state_data) do
  {:next_state, :player2_turn, state_data, {:reply, from, :ok}}
end

def player1_turn({:call, from}, :win, state_data) do
  {:next_state, :game_over, state_data, {:reply, from, :ok}}
end

def player1_turn({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :player1_turn}}
end

def player1_turn({:call, from}, _, _state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end

def player2_turn({:call, from}, {:guess_coordinate, :player2}, state_data) do
  {:next_state, :player1_turn, state_data, {:reply, from, :ok}}
end

def player2_turn({:call, from}, :win, state_data) do
  {:next_state, :game_over, state_data, {:reply, from, :ok}}
end

def player2_turn(_event, _caller_pid, state) do
  {:reply, {:error, :action_out_of_sequence}, :player2_turn, state}
end

def player2_turn({:call,from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :player2_turn}}
end

def game_over({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :game_over}}
end

def game_over({:call, from}, _, _state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end

defp set_islands_reply(from, state_data, status, status)
     when status == :islands_set do
  {:next_state, :player1_turn, state_data, {:reply, from, :ok}}
end
defp set_islands_reply(from, state_data, _, _) do
  {:keep_state, state_data, {:reply, from, :ok}}
end
```

```
end
```

Next we need to provide a default implementation for code_change/4. This callback specifies how we want the state and state data to change during a hot code reload. We won't be doing hot code reloads with Islands, so we'll return the state and state data as they are.

gen_statem/lib/rules.ex

```
def code_change(_vsn, state_name, state_data, _extra) do
  {:ok, state_name, state_data}
end
```

The terminate/3 function specifies any special cleanup we might need to do before the process terminates. We don't have any special needs here, so we can return anything. We'll just return the atom :nothing.

gen_statem/lib/rules.ex

```
def terminate(_reason, _state, _data), do: :nothing
```

With the missing callbacks defined, let's try starting a new process one more time.

```
$ iex -S mix
Erlang/OTP 19 [erts-8.2] [source] [64-bit] [smp:8:8] [async-threads:10]
              [hipe] [kernel-poll:false] [dtrace]
Compiling 1 file (.ex)
Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)
iex> alias IslandsEngine.Rules
IslandsEngine.Rules
iex> {:ok, pid} = Rules.start_link()
{:ok, #PID<0.124.0>}
```

That worked. We started the state machine and got a PID.

Now that we can start a new process, it's time to start doing something more interesting with it.

Adding New Behavior

When we're working with unfamiliar code, sometimes we need a little extra visibility to help us understand what's going on under the hood. A lot of what's happening in :gen_statem takes place in the Behaviour module itself, where we can't see it.

With both Agents and GenServers, we defined helper functions to provide more visibility. Let's do that again here to tell us the current state of the state machine. This will give us some insight into how our state machine is behaving as we develop it.

We'll need a client function first. Let's define it as show_current_state/1. The only argument it will need is the PID of the running state machine.

gen_statem/lib/rules.ex
```
def show_current_state(fsm) do
  :gen_statem.call(fsm, :show_current_state)
end
```

:gen_statem has call/2 and cast/2 functions for synchronous and asynchronous calls, just as GenServer does. We'll be using call/2 exclusively in this chapter because we'll always need to send a response back to the game server for each request.

We said that we need to name :gen_statem callback functions after a state, which means that each named callback will apply to only that state. The only state we currently have is :initialized which we set in init/1. Let's add that callback function now.

```
def initialized({:call, from}, :show_current_state, state_data) do
  {:next_state, :initialized, state_data, {:reply, from, :initialized}}
end
```

When :gen_statem triggers this callback, it provides all the arguments for us, just the way GenServer did. The first argument is a tuple indicating that this clause handles a call plus the pid of the caller. Then we get an atom representing the event as well as the state data.

The return tuple is really composed of two parts. The first three elements specify how to handle the state and state data. We're saying that the next state should be the same as the current state, :initialized, and that we're returning the state data as is. This part of the return is useful for the Behaviour itself.

If we had chosen another state to transition to, that state would magically have come into existence. We don't need to define the next state anywhere else for this to work. In our helper function, we just want to see the current state, not transition to a new one. Returning the current state as the next one prevents a state transition.

The second part of the return is a three element tuple representing the reply to the caller. The name of the current state is what we're here for. Out of that reply tuple, the caller will only receive the the current state, the atom :initialized.

:gen_statem is incredibly flexible in the way it lets us formulate replies. Let's take a look at a few equivalent ways of representing this same response.

We could remove that reply tuple and manually reply to the caller instead.

```
def initialized({:call, from}, :show_current_state, state_data) do
  :gen_statem.reply(from, :initialized)
  {:next_state, :initialized, state_data}
end
```

Since we are specifying that we don't want to transition to a new state, we could use the :keep_state atom instead of specifying a new state.

```
def initialized({:call, from}, :show_current_state, state_data) do
  {:keep_state, state_data, {:reply, from, :initialized}
end
```

And since we're not changing the state data either, we can use :keep_state_and_data instead.

gen_statem/lib/rules.ex

```
def initialized({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :initialized}}
end
```

Let's see how it works in a new console session.

```
iex> alias IslandsEngine.Rules
nil

iex> {:ok, pid} = Rules.start_link
{:ok, #PID<0.91.0>}

iex> Rules.show_current_state(pid)
:initialized
```

That's exactly what we want. We've just started the state machine, which leaves us in the :initialized state.

Now that we've had a taste of customizing our state machine, we're ready to go all in. To make it really useful, it will need to make decisions and answer questions on behalf of the game—Is it ok to add a new player now? Is it this player's turn to guess a coordinate?

Adding the ability to make those decisions is up next.

Fully Customizing Our State Machine

This is where we really make the state machine behave the way we need it to. We'll define a set of public functions that correspond to events in the system—setting islands, guessing a coordinate, etc. Then we'll add callbacks to decide if those events are permissible and send the appropriate responses. The responses will be either :ok or :error depending on what state the state machine is in.

We'll see just how flexible this combination of public functions and callbacks is. It will allow us to create state machines that fit the needs of any application we're working on.

Before we begin, it's helpful to have a picture of what we need to build, what the pieces look like, and how they fit together. Here's a representation of the state machine we need to implement including all the states and the direction of the transitions between them.

We're going to build our state machine one state at a time in the order they follow as the game progresses. For each state, we'll use the familiar OTP pattern—define a client function that wraps a :gen_statem module function that triggers a callback.

The difference is that we'll name the callback functions after the state they apply to. The callback's job will be to pattern match for the event and decide whether that event is permissible in that state.

With that picture in mind, let's work through the states the game passes through. We'll focus on the actions we'll allow in each state, and the events that trigger the transitions from one state to the next.

Initialized

State machines need a state to begin from, a way into the system of events and state transitions. From that beginning, incoming events will trigger state transitions—or not—depending on the rules we define within the state machine itself.

With :gen_statem, the beginning state comes from the return of init/1. In the case of Islands, a single player starts a game. This event puts the game in it's first state, :initialized. When we're in :initialized, the only permissible action is adding the second player.

We're going to focus on that one transition in this section, from :initialized to :players_set, as well as the event that triggers it, adding a second player.

Let's start with a client function for adding a second player.

gen_statem/lib/rules.ex
```
def add_player(fsm) do
  :gen_statem.call(fsm, :add_player)
end
```

Our state machine doesn't rely on any external state data. It keeps track of its own, and makes decisions accordingly. The only argument this function needs is the state machine's PID.

This function doesn't actually add another player. It makes a decision about whether it's ok to add another player based on the current state of the game. If it's ok, it returns :ok. If not, it returns :error.

The caller will need to get that response back, so we'll use :gen_statem.call/2. We pass it the PID as well as an atom representing the event, :add_player.

Now we need a callback. We want it to embody the rule that adding a player is permissible in the initialized state.

Because the rule is for the initialized state, we name the callback initialized/3. We pattern match for the {:call, from} tuple as the first argument, and the :add_player event as the second. :gen_statem also provides the state_data argument, which we defined in init/1.

gen_statem/lib/rules.ex
```
def initialized({:call, from}, :add_player, state_data) do
  {:next_state, :players_set, state_data, {:reply, from, :ok}}
end
```

The return tuple sends :ok back to the caller, meaning the event is permissible. It also specifies :players_set as the next state.

Let's see how it works in a new console session. We'll want to see that the state machine transitions correctly, so we'll need a new callback to show the current state when we're in :players_set.

gen_statem/lib/rules.ex
```
def players_set({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :players_set}}
end
```

OK, now we're ready to give this a try.

```
iex> alias IslandsEngine.Rules
nil

iex> {:ok, pid} = Rules.start_link
{:ok, #PID<0.91.0>}

iex> Rules.show_current_state(pid)
:initialized

iex> Rules.add_player(pid)
:ok

iex> Rules.show_current_state(pid)
:players_set
```

That's exactly what we want. Calling the add_player/1 function when we're in the :initialized state returns :ok and moves us into the :players_set state.

Adding a player is the only event we allow in the :initialized state. We need to return an error for any other event associated with that state. What we really need is a catchall clause to return an error for any events besides the ones we explicitly say are ok.

This rule is for the :initialized state, so we'll need another clause of the initialized/3 callback. If we use a variable like _event in the pattern match, it will match any other event.

gen_statem/lib/rules.ex

```
def initialized({:call, from}, _, _state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end
```

We're returning an error tuple to the caller because the event is not OK. Since it's an error, we don't want the state machine to transition to a new state. Returning the current state keeps us in :initialized.

Clause Order Matters

Since this catchall clause will always match, it's important to define it after all the other clauses of initialized/3. Otherwise, it will prevent any other initialized/3 clauses defined after it from ever matching.

We can try out the catchall clause in a new console session.

```
iex> alias IslandsEngine.Rules
nil

iex> {:ok, pid} = Rules.start_link
{:ok, #PID<0.104.0>}
```

```
iex> Rules.show_current_state(pid)
:initialized

iex> :gen_statem.call(pid, :some_random_event)
:error

iex> Rules.show_current_state(pid)
:initialized
```

That's just what we were looking for.

These two one-line functions hold a lot of power. Using pattern matching and multi-clause functions, they map a state to an event and decide whether the event is permissible. They're so simple, there are hardly any moving parts to break. If a problem does manage to creep in, there's almost no code to debug.

With that, we're ready to move on to the next state.

Players Set

There are times when we need more information than just the current state and an event to decide what to do. An instance of :gen_statem can save its own state data, just like an Agent or GenServer, to help with this. We'll see how in just a minute.

At this point, the second player has joined the game so we are in the :players_set state. In :players_set, the players can position their islands on the board. This translates to resetting their island coordinates. They can also set their islands, declaring their positions fixed for the rest of the game.

The transition we'll focus on here is from :players_set to :player1_turn. That's when the game really begins.

In the :players_set state, players can only move their islands and set them. Moving an island should not transition the game to a new state. Players can move their islands around as much as they want until they're set.

Each player can set their islands independently, at a different time from their opponent. If a single player has set their islands, this should not transition the game to a new state. When both players set their islands, though, the game should transition into :player1_turn.

Let's begin with moving islands.

We'll need a client function that takes the process PID and an atom representing the player that moved their island.

gen_statem/lib/rules.ex
```
def move_island(fsm, player) when is_atom player do
  :gen_statem.call(fsm, {:move_island, player})
end
```

There's a subtlety here that we need to capture. Players can move their islands at any time until they set them. Both players are almost certain to set their islands at different times. If player1 has set their islands but player2 hasn't, player1 should no longer be able to move their islands, but player2 should still be able to. While this condition exists, the state machine should remain in the :players_set state.

In other words, when in the :players_set state, the :gen_statem can have two different conditions:

- neither player has set their islands
- one player has set their islands and the other hasn't

The second player setting their islands is the event that triggers a state change.

In order to handle this properly, we need to keep track of whether each player has set their islands individually. We need to save this data in the state machine, and a struct is the right data structure because we'll only need two keys which will always be the same.

Let's define a new struct at the top of the IslandsEngine.Rules module. The keys will be :player1 and :player2. The default values will be :islands_not_set.

gen_statem/lib/rules.ex
```
defstruct player1: :islands_not_set, player2: :islands_not_set
```

We'll need to set the %Rules{} struct in the init/1 function as the initial data state instead of the empty list we had before.

gen_statem/lib/rules.ex
```
def init(:ok) do
  {:ok, :initialized, %Rules{}}
end
```

Now we can use the data in the %Rules{} struct to make the right decisions. :gen_statem passes the state data into the callback as the third argument. We get the player in the first argument tuple. With those and a case statement, we can craft a response.

gen_statem/lib/rules.ex
```
def players_set({:call, from}, {:move_island, player}, state_data) do
  case Map.get(state_data, player) do
    :islands_not_set ->
      {:keep_state_and_data, {:reply, from, :ok}}
    :islands_set ->
      {:keep_state_and_data, {:reply, from, :error}}
  end
end
```

If the value for the player key in the state is :islands_not_set, we return a three element tuple that says the action is OK. We keep the state machine in the :players_set state and don't change the state data.

```
{:keep_state_and_data, {:reply, from, :ok}}
```

If the value is :islands_set, we return the three element error tuple and keep the state machine in the :players_set state without changing the state data.

```
{:keep_state_and_data, {:reply, from, :error}}
```

Let's check this in a new console session. We'll start up a new state machine and add the second player to get into the :players_set state.

```
iex> alias IslandsEngine.Rules
nil
iex> {:ok, fsm} = Rules.start_link
{:ok, #PID<0.91.0>}
iex> Rules.add_player(fsm)
:ok
iex> Rules.show_current_state(fsm)
:players_set
```

Then we'll have player1 move their islands a couple of times and make sure the game stays in :players_set.

```
iex> Rules.move_island(fsm, :player1)
:ok
iex> Rules.show_current_state(fsm)
:players_set
iex> Rules.move_island(fsm, :player1)
:ok
iex> Rules.show_current_state(fsm)
:players_set
```

Great. That's what we want.

We still need to handle players setting their islands when the game is in :players_set. Let's work on that next.

We'll need a client function for this event. We'll call it set_islands/2. It will take the state machine's PID as well as a player atom.

gen_statem/lib/rules.ex
```
def set_islands(fsm, player) when is_atom player do
  :gen_statem.call(fsm, {:set_islands, player})
end
```

The caller will need a response, so we'll use :gen_statem.call/2. We'll pass it the PID and a tagged tuple with the event and the player as its two elements.

Now for the callback. We always allow a player to set their islands in :players_set, so the callback should always reply with a tuple containing :ok as the response.

The question is whether the callback should transition the state machine to :player1_turn. That should only happen if both players have set their islands.

gen_statem/lib/rules.ex
```
def players_set({:call, from}, {:set_islands, player}, state_data) do
  state_data = Map.put(state_data, player, :islands_set)
  set_islands_reply(from, state_data, state_data.player1, state_data.player2)
end
```

The first thing we do is update the %Rules{} struct to indicate that the player has set their islands. Then we call a private helper function to set the reply.

gen_statem/lib/rules.ex
```
defp set_islands_reply(from, state_data, status, status)
     when status == :islands_set do
  {:next_state, :player1_turn, state_data, {:reply, from, :ok}}
end
defp set_islands_reply(from, state_data, _, _) do
  {:keep_state, state_data, {:reply, from, :ok}}
end
```

The key to this function is the pattern match in the function head.

By naming two arguments exactly the same, we force their values to be the same. Variables can only bind to values once during a pattern match, even if they appear in different places in the pattern. The same variable cannot bind to two different values in the same match.

When we add a guard clause specifying the value we expect status to be, we're saying that both arguments need to equal that value. If the value of status is :islands_set, the game moves on to :player1_turn. If not, it stays in :players_set.

To complete our callback definitions, we'll also need a catchall clause of players_set/3 to return an error for any event other than the two we just specified.

gen_statem/lib/rules.ex
```
def players_set({:call, from}, _, state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end
```

This is nearly identical to the catchall clause we wrote for :initialized. The only differences are the callback name, and the value of the next state, :players_set.

Let's see how this works in the console. This time we'll go all the way through to the next state, so we'll need another clause that handles :show_current_state for :player1_turn.

gen_statem/lib/rules.ex
```
def player1_turn({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :player1_turn}}
end
```

After we start up a new state machine, we'll add a player to transition into :players_set.

```
iex> alias IslandsEngine.Rules
nil
iex> {:ok, pid} = Rules.start_link
{:ok, #PID<0.91.0>}
iex> Rules.add_player(pid)
:ok
iex> Rules.show_current_state(pid)
:players_set
```

Then we'll have each player move an island and show that the game is still in :players_set.

```
iex> Rules.move_island(pid, :player1)
:ok
iex> Rules.show_current_state(pid)
:players_set
iex> Rules.move_island(pid, :player2)
:ok
iex> Rules.show_current_state(pid)
:players_set
iex> Rules.move_island(pid, :player2)
:ok
iex> Rules.show_current_state(pid)
:players_set
```

Now we'll have both players set their islands. The game only transitions to :player1_turn after the second player sets their islands.

```
iex> Rules.set_islands(pid, :player2)
:ok
iex> Rules.show_current_state(pid)
:players_set
iex> Rules.set_islands(pid, :player1)
:ok
iex> Rules.show_current_state(pid)
:player1_turn
```

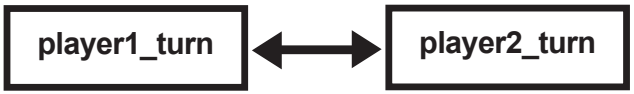
This behaves exactly as we want it to. Time to tackle the next state.

Player One's Turn

Not all state transitions are one-way. State machines often need to revisit previous states based on events in the system. Taking turns in a game is a perfect example of this. In a two person game, the state will transition back and forth between one player's turn and the other until one of them wins.

We're at the point where both players have set their islands, and the game is in :player1_turn. When it's the first player's turn, that player may guess a coordinate, and that player may win the game. No other events are permissible.

That's exactly what we'll focus on for our state machine now, the transition from :player1_turn and :player2_turn.



The client functions we'll need are guess_coordinate/2 and win/1.

guess_coordinate/2 takes the state machine's PID as well as a player atom. We don't need to keep track of which coordinate the player guessed. We just need to know if it's OK for the player to make a guess.

Since the caller will expect a response, we'll use :gen_statem.call/2. We'll pass it the PID as well as a tagged tuple with the event and the player.

gen_statem/lib/rules.ex

```
def guess_coordinate(fsm, player) when is_atom player do
  :gen_statem.call(fsm, {:guess_coordinate, player})
end
```

The callback only needs to determine if the player is :player1. If it is, we reply with :ok since it's player1's turn. Then we transition to :player2_turn. We have all the data we need in the function head to do the work.

gen_statem/lib/rules.ex
```
def player1_turn({:call, from}, {:guess_coordinate, :player1}, state_data) do
  {:next_state, :player2_turn, state_data, {:reply, from, :ok}}
end
```

This function clause won't match if the second argument isn't :player1. We were going to need a catchall clause to handle any events we won't allow anyway. Let's define one now.

This is nearly identical to the two other catchall clauses we've already defined.

gen_statem/lib/rules.ex
```
def player1_turn({:call, from}, _, _state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end
```

The last client function we'll need is win/1. It just takes the state machine's PID and passes the atom :win into :gen_statem.call/2.

gen_statem/lib/rules.ex
```
def win(fsm) do
  :gen_statem.call(fsm, :win)
end
```

The player1_turn/3 callback clause matching :win should always succeed. The Game will determine when a player wins. The callback's job is simply to transition the state to :game_over.

gen_statem/lib/rules.ex
```
def player1_turn({:call, from}, :win, state_data) do
  {:next_state, :game_over, state_data, {:reply, from, :ok}}
end
```

Let's try this out in the console to see how it works. Since we'll have the state machine transition to :player2_turn, we'll need a new clause to handle :show_current_state.

gen_statem/lib/rules.ex
```
def player2_turn({:call,from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :player2_turn}}
end
```

After we start a new process, we add a player and have both players set their islands to get us to the first players turn.

```
iex> alias IslandsEngine.Rules
nil
```

```
iex> {:ok, pid} = Rules.start_link
{:ok, #PID<0.91.0>}

iex> Rules.add_player(pid)
:ok

iex> Rules.set_islands(pid, :player1)
:ok

iex> Rules.set_islands(pid, :player2)
:ok

iex> Rules.show_current_state(pid)
:player1_turn
```

When :player1 guesses a coordinate, we should transition to :player2_turn.

```
iex> Rules.guess_coordinate(pid, :player1)
:ok

iex> Rules.show_current_state(pid)
:player2_turn
```

It works! Now let's build the transition back to :player1_turn.

Player Two's Turn

This state is the mirror image of :player1_turn. When it's the second player's turn, they can guess a coordinate or win the game. No other events are permissible. Much of this will seem familiar. We already defined the two client functions we'll need in this state, guess_coordinate/2 and win/1.

The callbacks for :player2_turn are nearly identical to those for :player1_turn. The only difference is that guessing a coordinate while in :player2_turn takes us back to :player1_turn.

gen_statem/lib/rules.ex

```
def player2_turn({:call, from}, {:guess_coordinate, :player2}, state_data) do
  {:next_state, :player1_turn, state_data, {:reply, from, :ok}}
end
```

The win callback is identical to the one we defined for :player1_turn.

gen_statem/lib/rules.ex

```
def player2_turn({:call, from}, :win, state_data) do
  {:next_state, :game_over, state_data, {:reply, from, :ok}}
end
```

As with all the other states, we'll need a catchall clause for player2_turn/3.

gen_statem/lib/rules.ex

```
def player2_turn(_event, _caller_pid, state) do
  {:reply, {:error, :action_out_of_sequence}, :player2_turn, state}
```

```
end
```

Now that we have these functions defined, let's move on to the last state of the game.

Game Over

State machines often, but not always, have an end state, one from which we can't transition. In Islands, we do have an end state, :game_over.

The game alternates between :player1_turn and :player2_turn until one player wins. When a player does win, the game transitions to the :game_over state. Once the game is over, neither player can take any further action. There's nothing else to do.

We won't need any new client functions. We will need a catchall clause for game_over/3 to return an error tuple if a player somehow tries to do something further.

gen_statem/lib/rules.ex

```
def game_over({:call, from}, _, _state_data) do
  {:keep_state_and_data, {:reply, from, :error}}
end
```

We'll need a clause that handles the :show_current_state action.

gen_statem/lib/rules.ex

```
def game_over({:call, from}, :show_current_state, _state_data) do
  {:keep_state_and_data, {:reply, from, :game_over}}
end
```

Now we can see our state machine make it through all the states.

Let's get a new one started and make sure it's in the :initialized state.

```
iex> alias IslandsEngine.Rules
nil
iex> {:ok, pid} = Rules.start_link
{:ok, #PID<0.91.0>}
```

```
iex> Rules.show_current_state(pid)
:initialized
```

Then we can add a player and make sure we transition to :players_set.

```
iex> Rules.add_player(pid)
:ok
iex> Rules.show_current_state(pid)
:players_set
```

Each player should be able to move an island and the state should still be :players_set.

```
iex> Rules.move_island(pid, :player1)
:ok
iex> Rules.move_island(pid, :player2)
:ok
iex> Rules.show_current_state(pid)
:players_set
```

When both players set their islands, the state should transition to :player1_turn.

```
iex> Rules.set_islands(pid, :player1)
:ok
iex> Rules.set_islands(pid, :player2)
:ok
iex> Rules.show_current_state(pid)
:player1_turn
```

Now the players should be able to alternate guessing coordinates, beginning with :player1.

```
iex> Rules.guess_coordinate(pid, :player1)
:ok
iex> Rules.show_current_state(pid)
:player2_turn
iex> Rules.guess_coordinate(pid, :player2)
:ok
iex> Rules.show_current_state(pid)
:player1_turn
```

When somebody wins, the state should become :game_over.

```
iex> Rules.win(pid)
:ok
iex> Rules.show_current_state(pid)
:game_over
```

That all looks fantastic.

With a handful of client functions and some callbacks—almost all of them one-liners—we've managed to encapsulate all the rules of Islands. We named the client functions after events in the system. With what we know about the interactions between client functions and callbacks, the code is self-documenting.

Most importantly, we've kept these rules completely separate from the application logic in our GenServer.

Now we're ready to make our state machine show it's true value. For each new game, we'll also start a new state machine. The game will communicate with that one state machine throughout the course of the game, asking questions and getting answers to whether given actions are permissible.

Integrate the :gen_statem with the GenServer

State machines are great, but they aren't very useful in isolation. Applications need to lean on state machines to answer questions about state, events, and transitions.

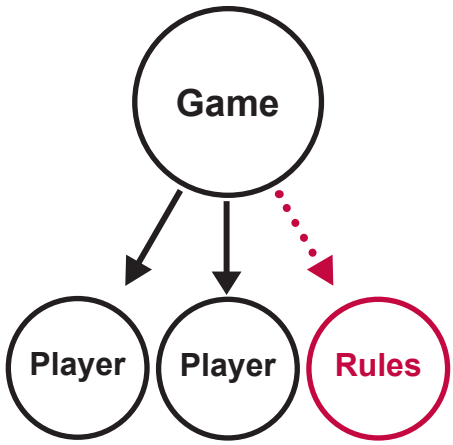
We need to get the game server talking to the state machine. Technically, we will integrate the :gen_statem into the game server. In practice, it's going to feel so much lighter than most integrations. We'll avoid tight coupling between the two processes. In fact, they'll remain two separate processes. They will only communicate via message passing.

This is where we're going to see the payoff for all our work with :gen_statem. Wiring it into our GenServer is going to be a snap. The replies we've set up from IslandsEngine.Rules are either :ok or :error. That will make it easy for us to use pattern matching and multi-clause functions to define different behavior.

Debugging will be easier as well. If there's a logic problem with the rules, there's one place to look for it. Maintenance will be easier. If we need to add more states and transitions, there's one place to do it. None of our GenServer code will need to change.

Our state machine runs as a separate process, just like the game engine. Each game will need its own state machine. The game will need to keep track of it and send messages to it. That's what we'll implement next.

Our plan will be to start a new instance of IslandsEngine.Rules each time we initialize a GenServer and set it in the GenServer's state data. That will make our top-level diagram for the game look like this:



First, we'll need to alias the IslandsEngine.Rules module in IslandsEngine.Game.

gen_statem/lib/game.ex
```
alias IslandsEngine.{Game, Player, Rules}
```

One easy way to make the state machine available is to set it in the struct we use for the game state. Since the state machine is a process that needs to be started at runtime, we can't set the actual PID here. We set it to :none first, just as we do for the players.

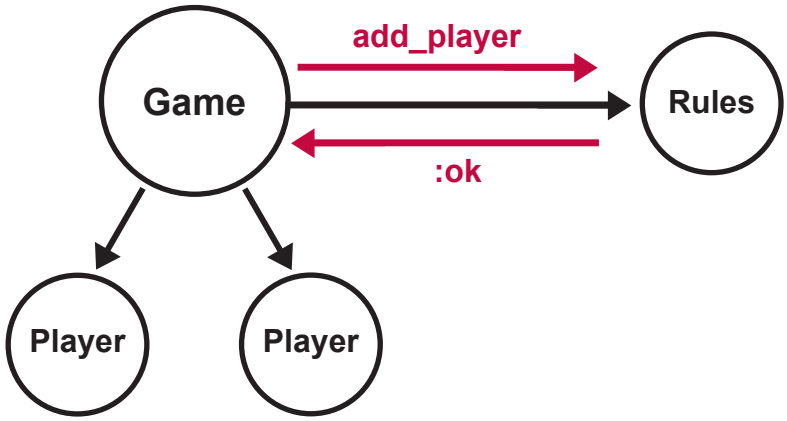
gen_statem/lib/game.ex
```
defstruct player1: :none, player2: :none, fsm: :none
```

In the init/1 function, we'll spawn a new state machine and set it in the game struct following the pattern we've already set.

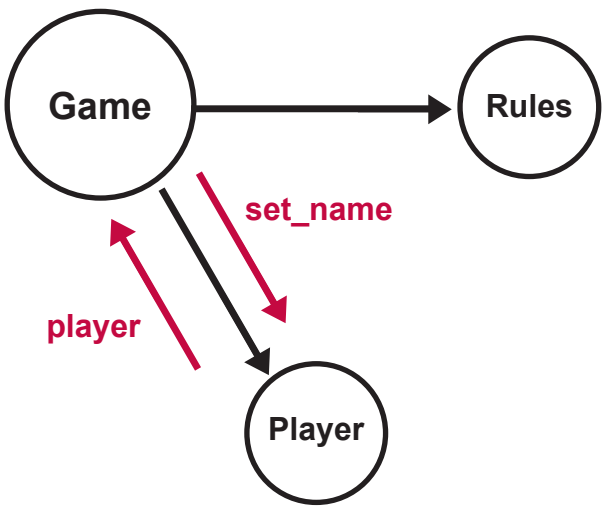
gen_statem/lib/game.ex
```
def init(name) do
  {:ok, player1} = Player.start_link(name)
  {:ok, player2} = Player.start_link()
  {:ok, fsm} = Rules.start_link
  {:ok, %Game{player1: player1, player2: player2, fsm: fsm}}
end
```

We're ready to enforce the rules for our first action, adding the second player.

We'll do this by first sending a message to the :gen_statem saying we want to add a second player.



Assuming the :gen_statem sends back :ok, we then send a message to the player module changing the name of the second player. The Player module will respond with a new %Player{} struct for the game state data.



We'll need to change the Game.add_player/2 a bit by calling Rules.add_player/1 and piping the response into a new private function that generates the correct reply.

gen_statem/lib/game.ex
```
def handle_call({:add_player, name}, _from, state) do
  Rules.add_player(state.fsm)
  |> add_player_reply(state, name)
end
```

We define two clauses of the add_player_reply/3 function. Each pattern matches on the response from the state machine. If the response is :ok, we add the new player's name to the game state and reply with :ok and the new state. If we get any other response, we pass that response through in our reply. We also include the unchanged game state.

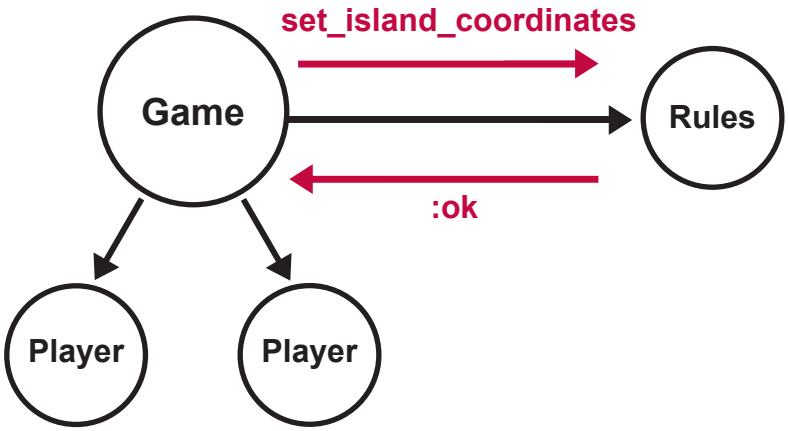
gen_statem/lib/game.ex
```
defp add_player_reply(:ok, state, name) do
  Player.set_name(state.player2, name)
  {:reply, :ok, state}
end
defp add_player_reply(reply, state, _name) do
  {:reply, reply, state}
end
```

The multiple definitions of add_player_reply/3 create a boundary that clearly separates the different behaviors we need. This helps us reason about our code by minimizing the lines of code inside each boundary.

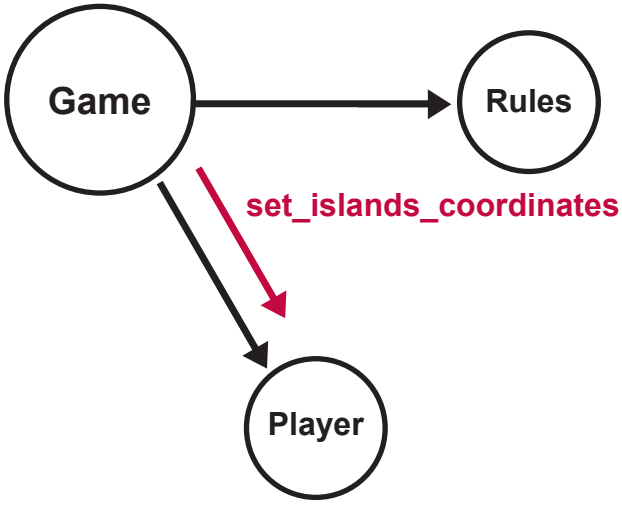
And that's it. That's all there is to it.

Let's tackle setting island coordinates next.

This follows the same pattern as adding a player. We tell the state machine that a player wants to move an island, and we pass the response off to a helper function to decide how to respond.



Assuming we get an :ok, we send a message to the Player module to update the island coordinates. That message will follow the same path we outlined when we wrote the game server.



Here's what that looks like updated in the set_island_coordinates callback.

gen_statem/lib/game.ex

```
def handle_call({:set_island_coordinates, player, island, coordinates}, _from, state) do
  Rules.move_island(state.fsm, player)
  |> set_island_coordinates_reply(player, island, coordinates, state)
```

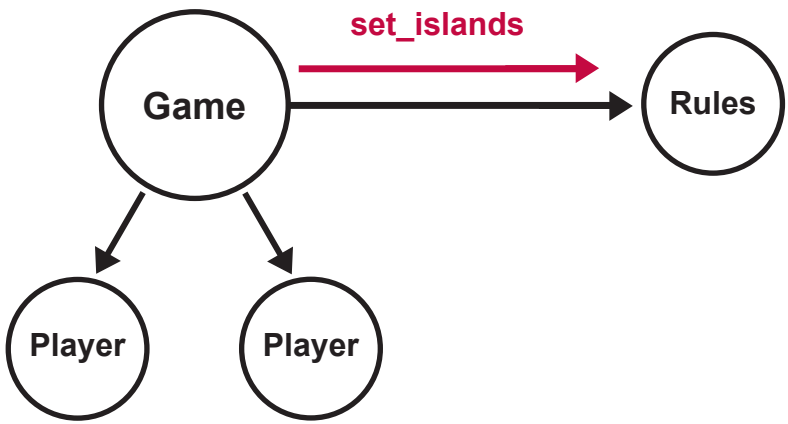
```
end
```

The helper function has two clauses, just like the one for adding a player. We've pushed the work of actually updating the island coordinates to the clause that matches :ok. There's no reason to do that work unless the the state machine says the action is OK.

gen_statem/lib/game.ex

```
defp set_island_coordinates_reply(:ok, player, island, coordinates, state) do
  Map.get(state, player)
  |> Player.set_island_coordinates(island, coordinates)
  {:reply, :ok, state}
end
defp set_island_coordinates_reply(reply, _player, _island, _coordinates, state) do
  {:reply, reply, state}
end
```

For setting islands, we're going to have to do a little more work. We don't have any code for this in the GenServer yet, so we'll need to write a client function/callback pair for it now. These will only need to talk to the state machine to update its state data. The game itself doesn't track this.



set_islands/2 takes the game server PID and a player atom. It passes the PID and a tuple of the event and the player into GenServer.call/2.

gen_statem/lib/game.ex

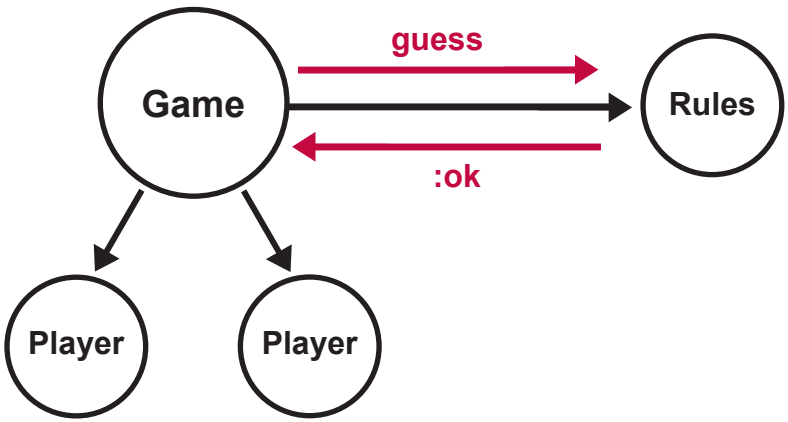
```
def set_islands(pid, player) when is_atom player do
  GenServer.call(pid, {:set_islands, player})
end
```

All the callback needs to do is pass along the reply from the state machine. Whether it is :ok or :error, we'll let the UI decide what to do with it later on.

gen_statem/lib/game.ex
```
def handle_call({:set_islands, player}, _from, state) do
  reply = Rules.set_islands(state.fsm, player)
  {:reply, reply, state}
end
```

Now we need to handle the key action in Islands, guessing coordinates.

We'll change the callback to check the rules, and write a helper function as we have with the other actions. That will need to happen before the forest check in the pipeline.



We'll take the response we get back from the :gen_statem, pass it to the helper, and then continue on with the rest of the pipeline we already have.

gen_statem/lib/game.ex
```
def handle_call({:guess, player, coordinate}, _from, state) do
  opponent = opponent(state, player)
  Rules.guess_coordinate(state.fsm, player)
  |> guess_reply(opponent.board, coordinate)
  |> forest_check(opponent, coordinate)
  |> win_check(opponent, state)
end
```

If the state machine replies with :ok, we go ahead with the guess and return the response. Otherwise, we reply with the :error. Either way, we keep moving through the pipeline.

gen_statem/lib/game.ex
```
defp guess_reply(:ok, opponent_board, coordinate) do
  Player.guess_coordinate(opponent_board, coordinate)
end
defp guess_reply({:error, :action_out_of_sequence}, _opponent_board, _coordinate) do
  {:error, :action_out_of_sequence}
end
```

Since we have a new possible argument from the helper, we'll need to add a clause to forest_check/2 to handle any :error responses.

gen_statem/lib/game.ex
```
defp forest_check(:miss, _opponent, _coordinate) do
  {:miss, :none}
end
defp forest_check(:hit, opponent, coordinate) do
  island_key = Player.forested_island(opponent, coordinate)
  {:hit, island_key}
end
defp forest_check({:error, :action_out_of_sequence}, _opponent_board, _coordinate) do
  {:error, :action_out_of_sequence}
end
```

We'll need to add a similar clause to win_check/2 for :error errors.

We'll also need to make one small change to win_check/2 itself. If the guess results in a win, we call Rules.win(state.fsm) to transition the state machine to :game_over.

gen_statem/lib/game.ex
```
defp win_check({hit_or_miss, :none}, _opponent, state) do
  {:reply, {hit_or_miss, :none, :no_win}, state}
end
defp win_check({:hit, island_key}, opponent, state) do
  win_status =
  case Player.win?(opponent) do
    true ->
      Rules.win(state.fsm)
      :win
    false -> :no_win
  end
  {:reply, {:hit, island_key, win_status}, state}
end
defp win_check({:error, :action_out_of_sequence}, _opponent, state) do
  {:reply, {:error, :action_out_of_sequence}, state}
end
```

That's it, we fully integrated the GenServer and :gen_statem. Each game now has it's own state machine to make decisions for it, and the game checks in with the state machine before every action.

Now we can take the complete game for a spin in the console.

Seeing the GenServer and :gen_statem in Action

We're at a great point right now. We've built all the behaviors the game needs. We've got a state machine to decide which actions we'll allow at any point in the game. The work we've done so far will let us walk through a whole game, following the rules. Let's do it!

We'll start by aliasing both the game and the state machine. Then we can start up a new game.

```
iex> alias IslandsEngine.{Game, Rules}
[IslandsEngine.Game, IslandsEngine.Rules]

iex> {:ok, game} = Game.start_link("Betty")
{:ok, #PID<0.124.0>}
```

We can use Game.call_demo/1 to get the game state.

```
iex> state = Game.call_demo(game)
%IslandsEngine.Game{fsm: #PID<0.341.0>, player1: #PID<0.232.0>,
 player2: #PID<0.340.0>}
```

Since the state machine is set in the game state, we have access to it as well. We can use it to show us which state we're in as we call game server functions.

```
iex> Rules.show_current_state(state.fsm)
:initialized
```

We can add a player through the game interface and see the state transition in the state machine.

```
iex> Game.add_player(game, "Barney")
:ok

iex> Rules.show_current_state(state.fsm)
:players_set
```

Now let's try to guess a coordinate before either player's islands are set. We should get the :error atom in response, and the state should stay the same.

```
iex> Game.guess_coordinate(game, :player1, :a1)
:error

iex> Rules.show_current_state(state.fsm)
:players_set
```

Both players can set island coordinates, and the state will still be :players_set.

```
iex> Game.set_island_coordinates(game, :player1, :dot, [:a1])
:ok
```

```
iex> Game.set_island_coordinates(game, :player2, :dot, [:b7])
:ok

iex> Rules.show_current_state(state.fsm)
:players_set
```

Let's stack the deck a little bit for player2. We'll have :player1 set all their islands to have the same coordinate, :a1. This will allow :player2 to win with a single guess. This will save us all a bit of tedium as we demonstrate winning the game.

```
iex> Game.set_island_coordinates(game, :player1, :atoll, [:a1])
:ok

iex> Game.set_island_coordinates(game, :player1, :l_shape, [:a1])
:ok

iex> Game.set_island_coordinates(game, :player1, :s_shape, [:a1])
:ok

iex> Game.set_island_coordinates(game, :player1, :square, [:a1])
:ok
```

When both players set their islands, the state transitions to :player1_turn.

```
iex> Game.set_islands(game, :player1)
:ok

iex> Game.set_islands(game, :player2)
:ok

iex> Rules.show_current_state(state.fsm)
:player1_turn
```

When player1 guesses a coordinate, we get the response back from the game. In this case, it's a miss. The state transitions to :player2_turn.

```
iex> Game.guess_coordinate(game, :player1, :d1)
{:miss, :none, :no_win}

iex> Rules.show_current_state(state.fsm)
:player2_turn
```

When player1 tries to guess out of turn, we see the :error atom. The state doesn't change.

```
iex> Game.guess_coordinate(game, :player1, :a1)
:error

iex> Rules.show_current_state(state.fsm)
:player2_turn
```

When player2 guesses the right coordinate, the game returns its own response. In this case, guessing a single correct coordinate wins the game. We see the :win, and the state transitions to :game_over.

```
iex> Game.guess_coordinate(game, :player2, :a1)
{:hit, :square, :win}

iex> Rules.show_current_state(state.fsm)
:game_over
```

That's exactly what we want.

We've been working on the game for a long time to get to this point. It's great to see it work!

Wrapping Up

We've done a lot of great work in this chapter. Islands now implements all the behavior it needs to. Our goal for this chapter was to bring in a little order, to enforce the rules. We have certainly accomplished that.

The state machine implementation we came up with is completely decoupled from the internals of the IslandsEngine.Game module. It can decide whether player actions follow the rules independently, without any knowledge of the game state.

Along the way, we learned a lot about finite state machines and Erlang's :gen_statem module in particular. We saw how we could map events to states in order to make decisions about application behavior. We learned how to manage state and state transitions in an Elixir application. We took a peek behind the curtain of Erlang Behaviours when we implemented missing callbacks for :gen_statem.

At this point, the game is nearly complete. The question we have to answer next is, what happens when things go wrong?

What we'll do in this chapter

- *look more deeply into the different ways of starting processes*
- *examine the different strategies for process supervision*
- *build a tree of supervisors for granularity*

CHAPTER 5

Process Supervision For Recovery

Content to be supplied later.

Part II

Add a Web Interface With Phoenix

Now that we have our game logic, it's time to provide a way to interact with that logic via the web. Phoenix does a fabulous job at this. We'll be generating a fresh Phoenix application and pulling in our game engine as a dependency. Since our game engine maintains state, we are in an ideal situation to make use of the persistent, multiplexed connections that Phoenix Channels provide.

Let's get to it!

What we'll do in this chapter

- *generate a new Phoenix application - without Ecto*
- *bring in our game engine as a dependency*
- *incorporate our game engine in the web interface's supervision tree*

CHAPTER 6

Generate a New Web Interface With Phoenix

Phoenix is a great web framework. It's fast, really fast. Its components are familiar and easy to work with. Phoenix is lightweight, modular, and explicit. There's almost no hidden magic. That's a big boost for maintainability.

Frameworks are nearly ubiquitous in web development today. For either the front end or backend, almost everyone uses some form of framework to build web applications.

There's good reason for this. Frameworks get us up and running quickly. They remove the need to re-implement common tasks for every project—routing, handling request parameters, and the like. Frameworks let us focus on our individual application's behavior instead of repetitive tasks.

The slippery slope is that frameworks make it all too easy to tangle the framework components and the application together in ways that really hurt us.

OTP Applications let us get around this in an elegant way. Phoenix itself is an OTP Application. The game engine we built in part one of the book is also an OTP Application.

Our task in part two of this book is to create a web interface with Phoenix for our game. We're going to use the Phoenix and game engine OTP Applications as building blocks to create a third Application that will keep the Phoenix interface separate from the game in a way that will make our job trivially easy.

Frameworks

Framework components represent what is common to all web applications. That's why the framework creators extracted them out into the framework. This is a great boon to developers because we don't need to solve the same problems over and over again. Things like routing requests to the right handler functions, getting the request parameters, handling response templates, setting cookies, the framework takes care of all that for us. The framework components make it easy interact with the business logic over the web. They make up the web interface for the application.

The business logic is unique to each application. This is the part that we can't extract into a common framework. It's what makes our application do interesting things, and gives it value. It's the most important part to us, because the success or failure of our application depends on how well this works.

But there's a serious, hidden-in-plain-sight problem here. We're so habituated to it that we hardly even notice.

Coupling

The way we normally build business logic with a framework is completely backwards. We create application behavior by adding more pieces of the framework—routes, controllers, models, and views. Each new model or controller we add contains a bit more logic. This mixes our business domain with the domain of the framework, and it couples the two inextricably and forever.

Why is that a big deal? We can't easily reuse the business logic with another interface. We can't test our business logic in isolation, outside the context of the framework code.

Let's say we wanted both a web interface and a Nerves device version of Islands. If we didn't have a separate OTP Application for Islands, we would need to completely re-implement the business logic for each interface.

Whenever we need to send an HTTP request to test a business rule, an alarm should go off at our workstation. Business rules should be completely separate from how we handle HTTP requests. Yet this is how we've been trained to test web applications.

This is why upgrading a framework to a new major version can sometimes be so painful. It's also why switching frameworks entirely seems like a herculean task. The way we normally work with frameworks makes this pain almost inevitable.

Phoenix is not Your Application

It's important to think about how we got into this situation, so we know how to get out of it.

The way we talk about web applications gets us in trouble right away. We say, "I'm building a Rails app." or an Ember app or a Phoenix app or an Elm app.

But that's not true. What we're really doing is building a chat app, or a banking app, or a game called Islands, with a Phoenix interface or an Ember interface or an Elm interface.

The problem is deeper than that though. There are a number of ways to look at it, but this resonates most with me. ORMs lead us directly into this coupling of business logic and framework components.

ActiveRecord models in Rails offer the clearest example of this, but the same idea applies across many frameworks. Let's say we are working in a domain in which one of the entities is a bicycle. We could begin modeling this with a plain Ruby class.

```
class Bicycle
  # We define bicycle-specific properties and behavior here.
end
```

We might define bicycle properties here like wheels, handlebars, pedals, and brakes. We could also define behaviors like pedaling, steering, and braking.

Rails tends to push us toward putting domain models, like out Bicycle class, in a database. ActiveRecord makes this very easy. We just have a model class inherit from ActiveRecord::Base.

```
class Bicycle < ActiveRecord::Base
  # Suddenly, bicycle behavior is mixed with database behavior.
end
```

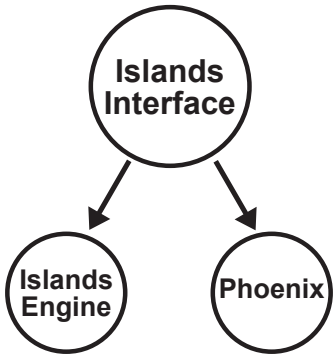
With this small change, everything is different. Our Bicycle class suddenly knows a lot more than just bicycle things. It knows how to connect to a database, read from and write to a table, validate data, perform transactions, generate queries, and a whole host of other things.

Our domain, in which a bicycle is just a bicycle, is suddenly entwined with the Rails domain, in which a Bicycle model is an interface to a database table. Once this happens, the two domains become glued together, and can't be separated without a rewrite.

Decoupling

That's not going to happen here. We already have our game logic separated out. Now we're going to layer on the web interface. The two will live happily side by side, and they won't be tightly coupled.

We built the core logic of the game as an OTP Application. That means we can bring it into any other OTP Application as a dependency, and all of its functionality will be available to us. Phoenix happens to be an OTP Application, which makes this job a snap. This one idea, this way of managing dependencies and building applications is quietly revolutionary. It's going to make the rest of our work with Phoenix seem trivial.



Of course, Erlang developers have been working in this quietly revolutionary way for a couple of decades now.

All our core logic needs is a web interface, and we'll use Phoenix to build one. Phoenix has all of the MVC components you're used to for those times when HTTP's request/response cycle fits best. It's also got a realtime, persistent connection layer called Channels built right in.

This is where all our hard work up to now is going to really pay off. From here, we'll be able to generate a fresh Phoenix application and bring our whole game in as a dependency. As we build out the interface, we won't be mixing in any application logic. We'll just call into the public interface of the game server that we've already built. There won't be any entanglement between the game and the interface.

OTP Applications are what allow us to build these separate, self-contained components. They're what allow us to compose them back together into larger applications as well. We'll explore them next.

OTP Applications

Despite the name, OTP Applications are not what we normally think of as software applications. They are reusable units of code that are bigger than modules. In fact, they most often contain multiple modules. They're similar in scale to libraries in other ecosystems. While they can function as libraries, they can also be so much more.

Applications can act as true building blocks for our programs, a means of putting together integral pieces of business logic to build a larger whole. Working with larger building blocks like these makes us really productive.

Applications can also stand on their own as what we traditionally think of as an application. The IslandsEngine application we developed in the first part of the book is one example. It is a fully functioning game just as it is, albeit with a pretty unfriendly user interface. As complete as it is, we can still use it as a building block for something larger, as we'll soon see.

:application is a specific OTP Behaviour written in Erlang, just like :gen_server and :gen_statem. There is a module in OTP which defines :application specific functions as well as a list of callbacks we need to implement. Elixir provides a wrapper module around the pure Erlang one called Application. We'll be using the Elixir wrapper most often in this chapter.

The Application Behaviour lets us do three things. It lets us define and name Applications. It facilitates dependency management among Applications. We can define hierarchies of Application dependencies, and the Behaviour will make sure they work correctly. The Behaviour also facilitates cleanly starting and stopping individual applications in a running BEAM.

"Cleanly" here means two things. It makes sure to start any dependent Applications before it starts itself. It also keeps track of any processes the Application spawns during startup or while it's running, and makes sure to stop them when the Application stops.

Now that we've got an idea of the significance of OTP Applications, let's dig a little deeper and see how they work.

Understanding OTP Applications

The good news is that we've been working with an OTP Application all along. At the beginning of the book, when we generated the brand-new IslandsEngine project, Mix automatically created it as an OTP Application. We didn't need to look deeply at the Application Behaviour then because IslandsEngine stood on its own for our purposes.

Now, though, we need to use it as a dependency, as a building block to create a larger project—the web interface we're going to build in this chapter. Understanding Application dependencies will clarify our work on this project, and any other Elixir projects we work on.

We already have examples of the Application Behaviour related files in IslandsEngine. We'll use them to understand dependency management as well as starting and stopping individual applications inside the BEAM. We'll see first hand the independence of OTP Applications that let's us solve the coupling problem so prevalent in web applications.

There are three parts to the implementation of an OTP Application, and Mix has a hand in all of them.

When we generate a project with mix new we get a file named mix.exs at the root of our project. mix.exs defines key aspects of the application, everything from its name and version number to a list of applications it depends on to build the project.

Mix also generates a Behaviour callback module in the /lib directory named after our project. In the case of our game engine, it generated /lib/islands_engine.ex. If we supply the --sup flag to mix new, the callback module will contain the start/2 callback function necessary to start the top level supervisor for the application. Without --sup, the file will be there, but it will be empty.

Once we compile the project, mix will generate an application resource file, written in Erlang, that the BEAM will use to work with our application.

Let's take a look at these files now starting with mix.exs.

Managing Dependencies

Any project's mix.exs file has two main functions—defining a project's metadata and managing its dependencies. Of the two, dependency management is by far the most common thing developers do, and it's the most important for our purposes as well.

Three functions defined in mix.exs do all the work for us. The project/0 function returns a keyword list of metadata about the application.

```
def project do
  [app: :islands_engine,
   version: "0.1.0",
   elixir: "~> 1.0",
   build_embedded: Mix.env == :prod,
   start_permanent: Mix.env == :prod,
   deps: deps()]
```

```
end
```

The app name, version number and Elixir version are pretty self-explanatory. build_embedded: makes sure that compilation artifacts end up in the _build directory. In this case, that would be true for the production environment. start_permanent: starts the system in such a way that the BEAM will crash if the top level supervisor crashes. This will be true for the production environment as well.

The deps: key holds a list of build-time dependencies this application depends on. The value here is the return value of the deps/0 function, also defined in mix.exs.

```
defp deps do
  []
end
```

IslandsEngine has no dependencies, so the return value here is an empty list. When we generate a new Phoenix application in the next section, we'll see an example with a number of dependencies.

There are actually two types of dependencies for OTP Applications, those that matter for runtime, and those that come into play for build/compile time. Mix uses the dependencies listed in the deps/0 function to build the project. Any application in this list can have its own dependencies. This is how we can compose a larger tree of dependencies, just as we saw with supervision trees in *the (as yet) unwritten chapter.design_for_recovery*.

The last function in mix.exs is application/0. It returns a keyword list of data related to starting the application. The value of the :extra_applications key is a list of application names which are the runtime dependencies. Mix will make sure these are running before it starts :islands_engine. :mod holds a tuple for the module name of the callback module as well as a list of options that the start/2 function in that module might need.

```
def application do
  [extra_applications: [:logger],
   mod: {IslandsEngine, []}]
end
```

IslandsEngine only depends on the :logger application at runtime. This dependency is a default for all OTP Applications Mix generates. Elixir itself supplies this, so we don't need to list it in the deps/0 function.

The reason that there are two different places to define dependencies is that it's possible to need a dependency for compilation, but not need it to be running inside the BEAM, and visa versa.

If our application doesn't have a supervision tree, for example if we omitted the --sup flag when we generated the project, we can omit the mod key completely.

```
def application do
  [extra_applications: [:logger]]
end
```

You might be thinking that this seems a little redundant. Shouldn't Mix be able to infer the application list from the deps list as long as we give it some clues? As of Mix 1.4, it can.

If the runtime dependencies are the same as the compile time ones, we can omit the :extra_applications key in application/0.

```
def application do
  [mod: {IslandsEngine, []}]
end
```

If there are runtime dependencies not listed in the deps/0 function, :logger for instance, we can handle that with the :extra_applications key.

```
def application do
  [extra_applications: [:logger],
   mod: {IslandsEngine, []}]
end
```

And if we have compile time dependencies that we don't need to start when we start our application, we can mark them as runtime: false in the deps/0 function.

```
defp deps do
  [{:some_new_dep, "> 0.0.0", runtime: false}]
end
```

That brings us to the end of dependency management in mix.exs. Once we have dependencies defined, we need to be able to start them inside the BEAM. That's where we're headed next.

Starting and Stopping OTP Applications

OTP Applications are so independent, we can start and stop them individually in the BEAM. When we start an individual application, OTP will start any necessary supervisor or worker processes along with it. When we stop that

same application, OTP makes sure to stop those supervisors and workers as well.

The work of defining which processes to start all comes together in the callbacks file. In the IslandsEngine project, that's /lib/islands_engine.ex.

Back in *the (as yet) unwritten chapter.design_for_recovery*, we took a good look at this file in the context of supervision trees. We won't need to go over it in much detail, but we should point out the use Application line that makes this module an Application Behaviour.

We should also quickly mention the start/2 function, which is there to start the top level supervisor for the application. Along the way, it's going to make sure that any child processes—supervisors or workers—get started as well.

```
defmodule IslandsEngine do
  use Application

  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    children = [
      supervisor(IslandsEngine.GameSupervisor, [])
    ]

    opts = [strategy: :one_for_one, name: IslandsEngine.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

When we compile the project, mix takes information from mix.exs and produces an application resource file, like this one at /_build/dev/lib/islands_engine/ebin/islands_engine.app.

This will always live in the /_build/dev/lib/<application_name>/ebin/ directory, and be named after our application with a .app file extension.

The contents of this file is what the BEAM needs in order to properly handle our application.

```
{application,islands_engine,
             [{registered,[]},
              {description,"islands_engine"},
              {vsn,"0.0.1"},
              {modules,['Elixir.IslandsEngine','Elixir.IslandsEngine.Board',
                        'Elixir.IslandsEngine.Coordinate',
                        'Elixir.IslandsEngine.Game',
                        'Elixir.IslandsEngine.GameSupervisor',
                        'Elixir.IslandsEngine.Island',
                        'Elixir.IslandsEngine.IslandSet',
                        'Elixir.IslandsEngine.Player',
```

```
                'Elixir.IslandsEngine.Rules']},
        {applications,[kernel,stdlib,elixir,logger]},
        {mod,{'Elixir.IslandsEngine',[]}}]}.
```

Now that we've taken a good look at all the pieces, lets take this out for a spin to see if we can learn more about how it behaves.

If you've ever wondered why running iex at the root of a project doesn't load that project and start the applications, but running iex -S mix does, we're about to see why.

The -S flag tells IEx to run a script before opening the shell. Notice that mix.exs has the .exs file extension, signifying that it's a script file. The mix part is short for mix run which will run a given script in the context of an application. The default script to run is mix.exs.

Running mix.exs triggers the Behavior callback function start/2 that we just saw in the callback module. That will start our application as well as any applications it depends on.

Let's go ahead and start a plain iex session from the root of our islands_engine project. This will be a generic IEx session that will not start :islands_engine.

:application exposes a handy function called which_applications/0 that will show us which OTP applications are currently running. This function isn't defined on Elixir's Application wrapper module, so we call it on the Erlang module instead. Let's see what it tells us.

```
iex> :application.which_applications
[{:logger, 'logger', '1.3.2'}, {:iex, 'iex', '1.3.2'},
 {:elixir, 'elixir', '1.3.2'}, {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

Each running application shows up as a three-tuple. Among the others, Elixir itself is an application. This is pretty much the bare minimum for running an IEx session.

Now let's quit out of that session and start a new one with iex -S mix, and then run :application.which_applications/0 again.

```
iex> :application.which_applications
[{:islands_engine, 'islands_engine', '0.0.1'}, {:logger, 'logger', '1.3.2'},
 {:mix, 'mix', '1.3.2'}, {:iex, 'iex', '1.3.2'}, {:elixir, 'elixir', '1.3.2'},
 {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

Great, now we see our :islands_engine application, and we also see :mix itself, since we implicitly invoked mix run.

If we start a new IEx session without starting the application, like this iex -S mix run --no-start, we'll see that the :islands_engine application isn't running, but :mix still is.

```
iex> :application.which_applications
[{:logger, 'logger', '1.3.2'}, {:mix, 'mix', '1.3.2'}, {:iex, 'iex', '1.3.2'},
 {:elixir, 'elixir', '1.3.2'}, {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

We can start :islands_engine manually with Application.start(:islands_engine).

```
iex> Application.start(:islands_engine)
:ok

iex> :application.which_applications
[{:islands_engine, 'islands_engine', '0.0.1'}, {:logger, 'logger', '1.3.2'},
 {:mix, 'mix', '1.3.2'}, {:iex, 'iex', '1.3.2'}, {:elixir, 'elixir', '1.3.2'},
 {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

If we try to start it again, we get an error saying that it's already started.

```
iex> :application.start(:islands_engine)
{:error, {:already_started, :islands_engine}}
```

We can also stop it with Application.stop(:islands_engine).

```
iex> Application.stop(:islands_engine)
:ok

20:55:42.291 [info]  Application islands_engine exited: :stopped
nil

iex> :application.which_applications
[{:logger, 'logger', '1.3.2'}, {:mix, 'mix', '1.3.2'}, {:iex, 'iex', '1.3.2'},
 {:elixir, 'elixir', '1.3.2'}, {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

Now let's see the Application Behaviour's runtime dependency management in action. While :islands_engine is stopped, let's also stop :logger, and then try to restart :islands_engine.

```
iex(2)> :application.stop(:logger)
:ok

iex(3)>
=INFO REPORT==== 20-Jan-2017::21:32:08 ===
    application: logger
    exited: stopped
    type: temporary

nil

iex> :application.which_applications
[{:mix, 'mix', '1.3.2'}, {:iex, 'iex', '1.3.2'}, {:elixir, 'elixir', '1.3.2'},
```

```
 {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

```
iex> :application.start(:islands_engine)
{:error, {:not_started, :logger}}
```

The Behaviour correctly remembered that :islands_engine depends on :logger and wouldn't start :islands_engine because :logger wasn't running.

:application exposes another handy function, ensure_all_started/1 which will behave the same as start/2, making sure all the runtime dependencies are running before trying to start the application we pass in.

```
iex> :application.ensure_all_started(:islands_engine)
{:ok, [:logger, :islands_engine]}
```

```
iex> :application.which_applications
[{:islands_engine, 'islands_engine', '0.0.1'}, {:logger, 'logger', '1.3.2'},
 {:mix, 'mix', '1.3.2'}, {:iex, 'iex', '1.3.2'}, {:elixir, 'elixir', '1.3.2'},
 {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

That's great. It's exactly what we expected to see.

Now that we've seen how OTP Applications work, we're ready to generate a new Phoenix project and bring IslandsEngine in as a dependency.

Generate a New Phoenix Application

This is where the fun really begins. With just a couple of shell commands we'll have a new web application up and serving pages, and Phoenix will serve them faster than you might have thought possible.

Before we start, make sure you have the Phoenix installer archive as well as Node.js installed on your system. Have a look at the *the (as yet) unwritten appendix.installation_instructions* if you need help installing them. Node.js is only necessary to manage front end dependencies, including Brunch, which Phoenix uses as a build tool. Node.js doesn't play a role within Phoenix proper.

By default, the Phoenix project generator will install Ecto, the Elixir data wrapping and query generating package. The generator will also install Brunch by default.

We won't need to use Ecto because we won't be using a database. We will use Brunch, though, because we'll need to serve some assets for the web version of our game.

We'll call the Phoenix interface application islands_interface. We'll pass that name into the phoenix.new mix task that the installer archive exposes as well as the --no-ecto flag telling phoenix.new not to install Ecto.

```
$ mix phoenix.new islands_interface --no-ecto
* creating islands_interface/config/config.exs
. . . # Creating lots more files here
* creating islands_interface/web/views/page_view.ex
```

We'll talk about many of these files and their purpose in the next chapter. For now, let's keep going with the project generation.

Once the project generator is done creating files, it will ask if we want to install the application dependencies. We should say yes by either typing a "y" or just hitting return.

```
Fetch and install dependencies? [Yn] y
* running mix deps.get
* running npm install && node node_modules/brunch/bin/brunch build
```

Before we move on, let's take a quick look at the functions in mix.exs that the project generator created. We'll start with the project/0.

```
def project do
  [app: :islands_interface,
   version: "0.0.1",
   elixir: "~> 1.3",
   elixirc_paths: elixirc_paths(Mix.env),
   compilers: [:phoenix, :gettext] ++ Mix.compilers,
   build_embedded: Mix.env == :prod,
   start_permanent: Mix.env == :prod,
   deps: deps()]
end
```

The most important thing to notice here is the name of our application. It's :islands_interface, not :phoenix. The project generator didn't create a "Phoenix application", it created a new OTP Application for us with it's own identity, above Phoenix itself.

The deps/0 function tells us more.

```
defp deps do
  [{:phoenix, "~> 1.2.1"},
   {:phoenix_pubsub, "~> 1.0"},
   {:phoenix_html, "~> 2.6"},
   {:phoenix_live_reload, "~> 1.0", only: :dev},
   {:gettext, "~> 0.11"},
   {:cowboy, "~> 1.0"}]
end
```

Our :islands_interface application brings in :phoenix, the OTP Application, as well as other Phoenix related applications as dependencies. :islands_interface is the root node of a tree of dependencies, and these others are the next layer of nodes down.

The application/0 function doesn't hold any surprises. It makes sure that we start all the dependencies listed in deps/0.

```
def application do
  [mod: {IslandsInterface, []},
   applications: [:phoenix, :phoenix_pubsub, :phoenix_html,
                  :cowboy, :logger, :gettext]]
end
```

Ok, let's get back to the installation process. After mix fetched all our project's dependencies, it told us what our next steps should be.

```
We are all set! Run your Phoenix application:

    $ cd islands_interface
    $ mix phoenix.server

You can also run your app inside IEx (Interactive Elixir) as:

    $ iex -S mix phoenix.server
```

Let's follow those directions now. Change into the project directory and start the server with mix phoenix.server.

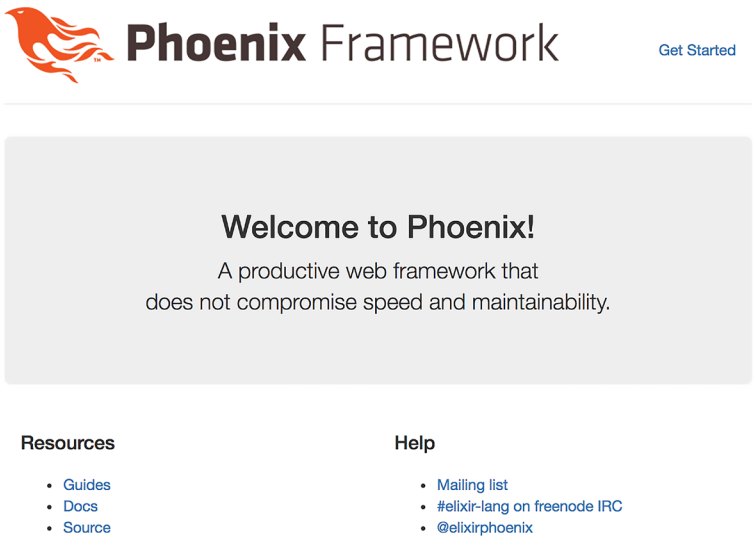
```
$ cd islands_interface/
$ mix phoenix.server
==> fs (compile)
Compiled src/sys/inotifywait_win32.erl

. . . # Lots of compilation here

==> islands_interface
Compiling 11 files (.ex)
Generated islands_interface app
[info] Running IslandsInterface.Endpoint with Cowboy using http on port 4000
22 Jul 09:12:05 - info: compiled 5 files into 2 files, copied 3 in 1.2 sec
```

By executing mix phoenix.server, we trigger the initial compilation of all the Elixir files in the project. That will generate the application resource file at /_build/dev/lib/islands_interface/ebin/islands_interface.app.

You might also see a warning about a new hex version that's available. Feel free to follow the instructions for upgrading.

```
A new Hex version is available (0.12.1), please update with `mix local.hex`
```

Now for the really exciting part. The very last bit of the compilation message let us know that the Erlang web server Cowboy is running our application on port 4000. Let's point a web browser at http://localhost:4000 and see what we get.



Just for fun, go ahead and reload the welcome screen a few times and take a look at the terminal screen that you started the application from.

```
[info] GET /
[debug] Processing by IslandsInterface.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 263µs
[info] GET /
[debug] Processing by IslandsInterface.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 224µs
[info] GET /
[debug] Processing by IslandsInterface.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 186µs
[info] GET /
[debug] Processing by IslandsInterface.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 197µs
```

Yes, with Phoenix, we can measure the page load times in *microseconds*.

We're ready to perform the quietly revolutionary act we talked about at the beginning of this chapter. We're going to bring in the logic for Islands as both a build time and runtime dependency for our new application.

Adding a New Dependency

Now that we have a new application to act as the web interface, we need to bring the game engine in as a dependency. The interface needs to have access to all the game logic we wrote in the first part of the book. In particular, it needs to be able to see all the public functions in the IslandsEngine.Game module.

This works just like any other dependency in Elixir, and it is about as easy as it can possibly be. There are only two steps involved.

We'll need to compile IslandsEngine in with the rest of the project as well as start it when we start the IslandsInterface project.

To make :islands_engine a compile time dependency, we'll add it to the deps/0 function in islands_interface/mix.exs.

```
defp deps do
  [{:phoenix, "~> 1.2.1"},
   {:phoenix_pubsub, "~> 1.0"},
   {:phoenix_html, "~> 2.6"},
   {:phoenix_live_reload, "~> 1.0", only: :dev},
   {:gettext, "~> 0.11"},
   {:cowboy, "~> 1.0"},
   {:islands_engine, path: "../islands_engine"}]
end
```

Notice that we used a path dependency for :islands_engine. That allows us to provide the pathname to a project on the local filesystem, and the Elixir package manager, Hex, will take care of the rest.

To make it a runtime dependency, we'll add it to the application/0 function, also in islands_interface/mix.exs.

```
def application do
  [mod: {IslandsInterface, []},
   applications: [:phoenix, :phoenix_html, :cowboy, :logger,
                  :gettext, :islands_engine]]
end
```

That's it. We can give this a try in the console by running iex -S mix phoenix.server at the root of the IslandsInterface project.

The first thing we should do is see which applications are running with :application.which_applications/0.

```
iex> :application.which_applications()
[{:islands_interface, 'islands_interface', '0.0.1'},
 {:islands_engine, 'islands_engine', '0.0.1'},
 {:cowboy, 'Small, fast, modular HTTP server.', '1.0.4'},
 {:cowlib, 'Support library for manipulating Web protocols.', '1.0.2'},
 {:ranch, 'Socket acceptor pool for TCP protocols.', '1.2.1'},
 {:phoenix_html,
  'Phoenix.HTML functions for working with HTML strings and templates',
  '2.9.2'},
 {:phoenix_pubsub, 'Distributed PubSub and Presence platform\n', '1.0.1'},
 {:logger, 'logger', '1.3.2'},
 {:gettext, 'Internationalization and localization through gettext', '0.13.0'},
 {:phoenix,
  'Productive. Reliable. Fast. A productive web framework that
              does not compromise speed and maintainability.',
  '1.2.1'}, {:eex, 'eex', '1.3.2'},
 {:poison, 'An incredibly fast, pure Elixir JSON library', '2.2.0'},
 {:plug,
  'A specification and conveniences for composable modules
   between web applications',
  '1.3.0'}, {:mime, 'A MIME type module for Elixir', '1.0.1'},
 {:hex, 'hex', '0.15.0'}, {:inets, 'INETS  CXC 138 49', '6.3.1'},
 {:ssl, 'Erlang/OTP SSL application', '8.0'},
 {:public_key, 'Public key infrastructure', '1.2'},
 {:asn1, 'The Erlang ASN1 compiler version 4.0.3', '4.0.3'},
 {:crypto, 'CRYPTO', '3.7'}, {:mix, 'mix', '1.3.2'}, {:iex, 'iex', '1.3.2'},
 {:elixir, 'elixir', '1.3.2'}, {:compiler, 'ERTS  CXC 138 10', '7.0.1'},
 {:stdlib, 'ERTS  CXC 138 10', '3.0.1'}, {:kernel, 'ERTS  CXC 138 10', '5.0'}]
```

That's a lot more applications running than we had with :islands_engine. The main things to note are that both :islands_engine and :islands_interface started. Also note that Phoenix itself is listed as a started application.

Since both the :islands_engine and :islands_interface applications are available, let's see if we can start a new game from the console using the public interface of IslandsEngine.Game.

```
iex> IslandsEngine.Game.start_link "Betty"
{:ok, #PID<0.465.0>}
```

That's perfect. We can start up a new game from within the Phoenix interface. Once we have the game's PID, we can use it to call any of the public IslandsEngine.Game functions.

With :application.which_applications/0 we can see which applications are running at any given time, but we don't get a sense of the structure, the tree of applications and their dependencies.

Mix gives us a tool to do this, the deps.tree task. Let's run it at the root of the IslandsInterface project.

```
$ mix deps.tree
islands_interface
├── gettext ~> 0.11 (Hex package)
├── islands_engine (../islands_engine)
├── phoenix_pubsub ~> 1.0 (Hex package)
├── cowboy ~> 1.0 (Hex package)
│   ├── cowlib ~> 1.0.0 (Hex package)
│   └── ranch ~> 1.0 (Hex package)
├── phoenix_html ~> 2.6 (Hex package)
│   └── plug ~> 1.0 (Hex package)
│       ├── cowboy ~> 1.0.1 or ~> 1.1 (Hex package)
│       └── mime ~> 1.0 (Hex package)
├── phoenix ~> 1.2.1 (Hex package)
│   ├── cowboy ~> 1.0 (Hex package)
│   ├── phoenix_pubsub ~> 1.0 (Hex package)
│   ├── plug ~> 1.1 (Hex package)
│   └── poison ~> 1.5 or ~> 2.0 (Hex package)
└── phoenix_live_reload ~> 1.0 (Hex package)
    ├── fs ~> 0.9.1 (Hex package)
    └── phoenix ~> 1.0 or ~> 1.2-rc (Hex package)
```

This confirms what we suspected, that :islands_interface is at the root of this tree. The important thing to notice is that both :islands_engine and :phoenix are parallel and equal in this tree. Both live *under* :islands_interface, and both provide functionality to make the whole web application work.

This is a subtle but critical point. It's what allows us to let the interface talk to the game logic's public interface, and keeps the two nicely decoupled.

That's the theory, now let's prove it.

Call the Logic from the Interface

We've seen that we can start a new game from within an IEx session begun with iex -S mix phoenix.server. That means means we can call the public functions of the IslandsEngine.Game module from any Phoenix component.

At the beginning of this chapter, we talked a lot about decoupling the interface from the business logic. We made a bold claim and said that this would make our work trivially easy. Throughout the chapter, we've shown how we can keep the two separated, but included in a common project. What we haven't

yet shown is how those two will communicate. It's time to back that claim up.

We're going to walk through this pretty quickly. We'll be working with a few new files, but we won't be spending a lot of time explaining them. Don't worry, we'll take a much closer look at them in the next chapter.

The good news is that Phoenix provides a full, working example of all the files we'll need. It's the welcome page we saw when we started the server for the first time. We're going to modify some of those files to perform an experiment.

The first thing we'll do is create a form in the index template /islands_interface/web/templates/page/index.html.eex. We'll use the form_tag/2 and tag/2 functions from the :phoenix_html application. You'll recall that we saw :phoenix_html in our list of dependencies from mix.exs.

```
<div class="jumbotron">
  <h2><%= gettext "Welcome to %{name}", name: "Phoenix!" %></h2>
  <p class="lead">A productive web framework that<br />does not compromise speed and maintainabi
  <p>
    <%= form_tag("/test") do
      [tag(:input, type: "text", name: "name"),
      tag(:input, type: "submit", value: "New Game")]
    end%>
  </p>
</div>
```

With form_tag/2, the default action is POST, and we pass it the route /test. The tag/2 functions create a text input for a player's name and a submit button.

We just told that form to post all requests to /test, which is a route we don't currently have. Let's add it now in islands_interface/web/router.ex.

```
scope "/", IslandsInterface do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
  post "/test", PageController, :test
end
```

That route says that any POST request to /test should be handled by the test function in the IslandsInterface.PageController module.

That module exists, but the function doesn't, so let's add it now to islands_interface/web/controllers/page_controller.ex.

```
defmodule IslandsInterface.PageController do
  use IslandsInterface.Web, :controller

  def index(conn, _params) do
    render conn, "index.html"
```

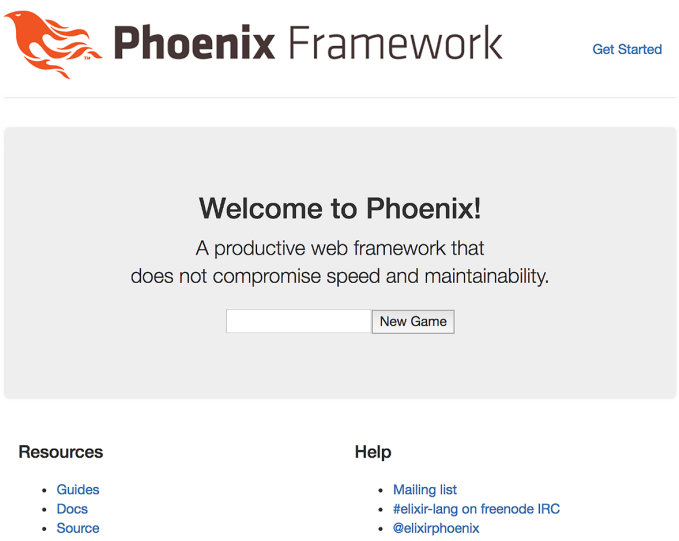
```
  end

  def test(conn, %{"name" => name}) do
    {:ok, _pid} = IslandsEngine.Game.start_link(name)
    conn
    |> put_flash(:info, "You entered the name: " <> name)
    |> render("index.html")
  end
end
```

In the function head, we pattern match on `%{"name" => name}`. This second argument is the incoming parameters map from the request. The match binds the value of the `"name"` key to the `name` variable, and makes it available inside the body of the `test` function.

Notice that we're pattern matching for a successful start of the game server `{:ok, _pid} = Supervisor.start_child(:game_supervisor, [name])`. If the game server fails to start, we'll get an error page.

We're also setting a flash message that will let us know which name we entered into the form, and then we're re-rendering the same index page we were just on.

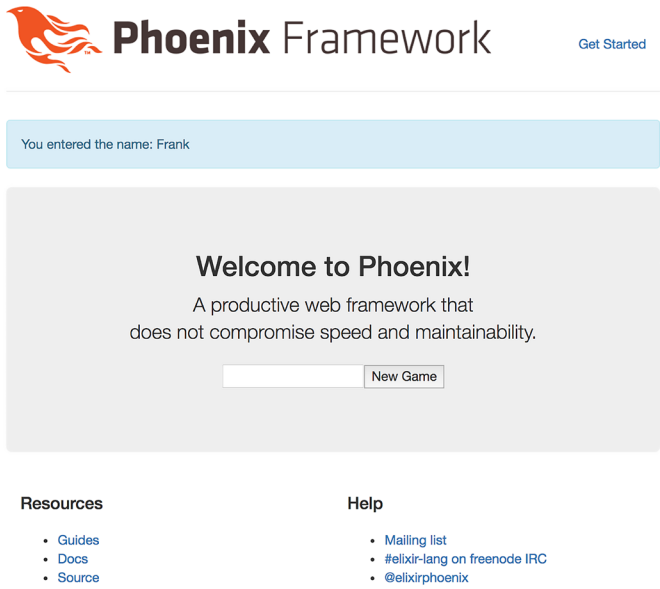
Go ahead and start the server from the root of the IslandsInterface project with `$ iex -S mix phoenix.server`. Then head over to `http://localhost:4000/` with your favorite browser. The form we just put in the `index.html.eex` file should now be there.



Before we do anything, let's make sure that the game supervisor hasn't started any games yet.

```
iex> Supervisor.which_children(:game_supervisor)
[]
```

Now go ahead and add a name to the input field and hit submit. That should take you right back to the welcome page, and you should see the name you entered at the top of the page in a flash message.



At this point, you can go back to the terminal window to check for the POST request you should have gotten by submitting the form.

```
iex> [info] POST /test
[debug] Processing by IslandsInterface.PageController.test/2
  Parameters: %{"_csrf_token" => "<long_string_omitted>",
                "_utf8" => "✓", "name" => "Frank"}
  Pipelines: [:browser]
[info] Sent 200 in 55ms
```

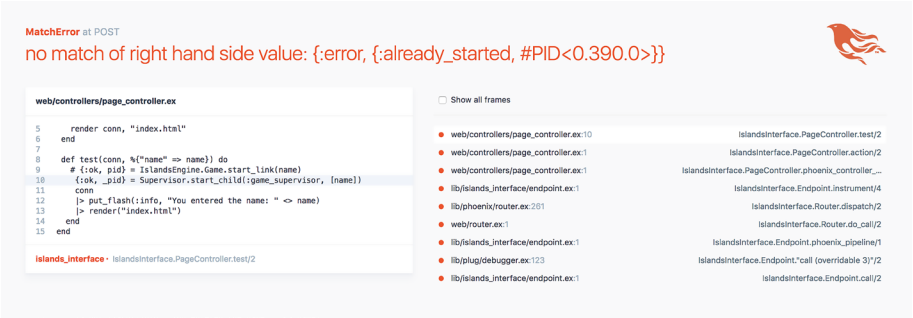
Great, that looks like just what we want.

Let's check to see that the game supervisor actually started a new game process.

```
iex> Supervisor.which_children(:game_supervisor)
[{:undefined, #PID<0.361.0>, :worker, [IslandsEngine.Game]}]
```

And we can see that it did. Excellent.

For fun, try entering exactly the same name into the form and hitting submit again. You should get an error page telling you that the game server was already started. Recall that we give each game server a global name based on the string we pass in to start the game server, and we can only start one server at a time with that name.



That's exactly the error we get.

Now that we can start a game server, and we know that the IslandsEngine.Game module is available inside Phoenix components in IslandsInterface, the world is our oyster. We can call any of the game server's public functions from any module in IslandsInterface. This is what will allow us to play Islands on the web.

Wrapping Up

We've done a lot in this chapter in a short amount of time. We created a new project for our web interface. We brought the game logic in as a dependency, and we got the interface to call into the game server.

That's the surface level view. Looking at it more deeply, we've solved one of the most vexing problems related to using web frameworks. We've created a clean separation between logic and interface that will make testing and maintaining our application a breeze. If we ever need to upgrade to a newer major version of Phoenix, it'll be a much, much easier task than it would be with other frameworks.

At this point, we're ready to tackle one of the most exciting parts of Phoenix—Channels. Channels provide persistent, stateful connections between stateful backends and frontends, and they scale beautifully. That's where we're headed next.

What we'll do in this chapter

- *create a channel which communicates directly with a GenServer*
- *use the topic:subtopic convention to focus communication on a single GenServer*
- *define separate handle_in events for each game command—new game, join game, fire shot, and so on*
- *interact with our channel in the console to see it work*

CHAPTER 7

Create Persistent Connections With Phoenix Channels

Phoenix Channels are just amazing. They really are Phoenix's killer feature. They provide persistent connections between stateful servers and stateful clients. They're incredibly fast, and they can truly scale.

Channels allow us to fulfill the promise we made in the very beginning of this book, to connect a stateful backend to a stateful front end with a persistent, stateful connection.

We're going to build a channel that will allow us to directly interact with the game engine. The channel callback functions we define will match the public interface of the game. We'll build new functionality with callbacks in our own new channel module, and then we'll exercise those callbacks in two browser window consoles to mimic two players playing the game.

The Beauty of Channels

Channels fundamentally change the nature of what we're able to do on the web.

Channels scale incredibly well. In one test on a powerful machine, the Phoenix team was able to establish two million simultaneous channel connections. These weren't just static connections. The team was able to broadcast messages to all two million clients within a few seconds.

Let that sink in for a minute. Think about what your application would look like if every user of your system had their own persistent connection to the server.

Channels are soft-realtime communication conduits. Clients join a channel on a specific topic. That's the same as saying that clients subscribe to a topic on a channel. Then they're able to facilitate conversations on those topics. Channels are multiplexed, so a single channel can support bi-directional messages on many topics.

Most often, the client is a front end web client, but it could be anything that knows how to send a message to a channel, including another server.

It's tempting to think of channels as equivalent to raw websockets, the most common protocol that channels use for transporting data. The reality is that channels offer a lot of nice features over websockets alone. Channels can transport data over a number of different protocols, including custom ones you can write yourself. By default, they'll use websockets, and fall back to longpolling if websockets aren't available.

Channels also handle failure well. When networks loose connection, channels know how to re-establish communication and carry on where they left off. That's the kind of thing you need to build yourself if you're using websockets alone.

Channels require less code to implement than traditional MVC components. There's a single module to write callbacks in instead of a controller, view, template, and schema.

The key to our game channel implementation is that there will be no business logic in the channel. All the channel callbacks will simply call directly into the game engine, pattern match on the response, and determine the correct reply to send back to the client.

Another way to look at this is that the channel will be concerned only with the behavior that is appropriate for this layer—determining which response to reply with and figuring out which clients to send that response to.

Before we move on to implementing our own channel, let's take a look at the moving parts that make up a channel.

The Pieces That Make a Channel

We often talk about Phoenix Channels as if each one is a single, monolithic entity that works on its own. In fact, there are a number of moving parts

acting in harmony across multiple layers that make channels work as well as they do.

Let's take a quick look at the most important ones to get a better feeling for the whole.

The Channel Module

The channel module is the tip of the iceberg, the visible part that we will interact with the most. It's a custom Behaviour defined within Phoenix. The Behaviour specifies that we define a `join/3` callback for allowing clients to join a specific topic as well as multiple `handle_in/3` callbacks to match, handle, and respond to messages sent from clients.

Socket

Phoenix.Socket is also a custom Behaviour defined in the Phoenix application. It is responsible for establishing and maintaining the connection between clients and a channel. The socket also keeps track of which transport method the channel uses.

Socket is also a struct used to define and hold the state of the connection. It's analogous to the connection struct in the stateless MVC parts of Phoenix.

Transport

Channels rely on protocols to move messages between the client and the server. That's what the transport layer is for. Phoenix ships with two types of transports built in: websockets, and longpolling.

Socket.Transport is also an API for building transports. It's possible to create your own custom transport layer for whatever protocol you like by following the Socket.Transport API.

Phoenix PubSub

Channels are very flexible in the way they allow us to route messages from the channel to clients. That's handled by a separate package called `phoenix_pubsub`. Pubsub is short for "publish and subscribe" it's a way for clients to register with a channel (subscribe) in order to get sent published messages. The way clients subscribe is via the `join/3` callback function.

Presence

Phoenix presence is a really incredible piece of technology. It uses data types from cutting edge computer science research—CRDTs, conflict-free replicated datatypes—in a web framework you can use right now.

Presence solves the hardest and most extreme edge cases in keeping track of clients in channels—multiple nodes in a distributed cluster, clients connected on multiple devices, and anything going wrong with either the network or the clients. Presence solves these problems in a mathematically provable way.

Not only does Presence solve the original problem of channel membership, it promises to be useful for service discovery, process discovery, and anything that we need to track on BEAM nodes distributed across a network.

Client Code

Channels on the server are only half the story. We need clients to complete the picture. Channel client packages exist for a number of different languages and platforms. We'll use the JavaScript client that ships with Phoenix as we write functions directly in a browser window's JavaScript console.

That's the lay of the land. We've got all the info we need to get started, so we might as well dive right in. We're going to implement the channel as well as the JavaScript necessary for the front end to talk to it.

Let's Build It

The big picture for this final chapter is that we're opening up the public interface we created for the game server to the web. The game server is a stateful system. Modern web frontends are also stateful. We're going to build a Phoenix Channel as a stateful, persistent connection between these two stateful systems. This new channel's public interface will become the mechanism clients will use to interact with the game.

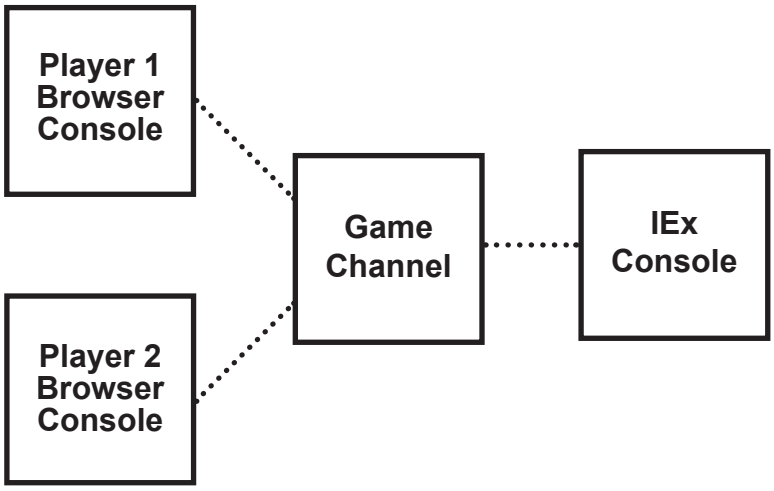
Here's the Plan

We're going to build that channel function by function, and we'll check its behavior at each step along the way by calling JavaScript code in the browser's developer tools console. We want the new channel to communicate directly to an individual GenServer process for a game. Since channels multiplex messages, a single channel can handle clients sending messages to many topics. Which is to say that a single channel can handle the communication for many games. Checking that this communication works means that we'll

need to be able to see what's happening from the browser all the way down to the server.

To fully exercise this, we're going to use two separate browser windows. That will mimic two players playing the game on the web. We'll work with the developer tools JavaScript console in each browser window We'll also have an IEx session open to check what's happening to the state of the game server.

That setup is going to look like this.



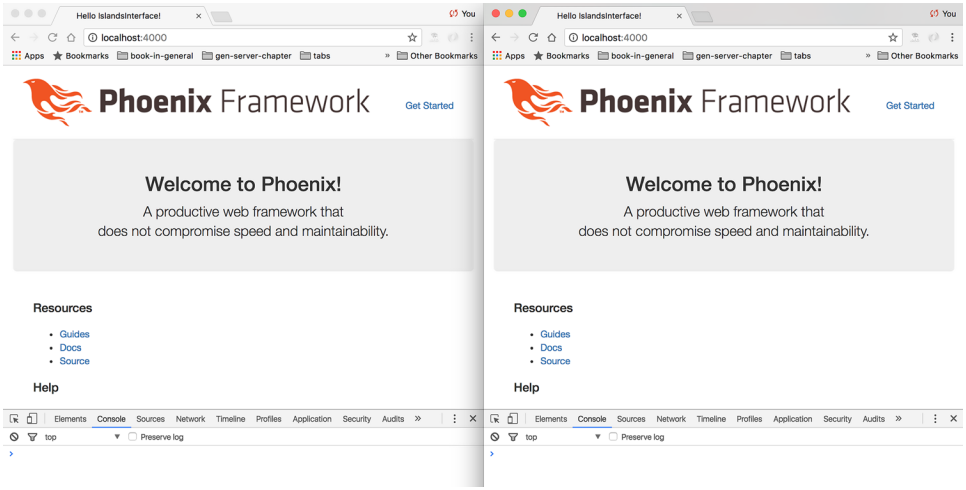
There are a lot of moving parts here, and at times it might seem like building a model ship in a bottle. I'm going to err on the side of clarity over brevity to make this easier to follow.

To help with clarity, let's stick with a convention. We'll keep the two browser windows open side by side, and we'll say that the window on the left represents player1, and the window on the right is player2.

Go ahead and run `iex -S mix phoenix.server` at the root of the `islands_interface` project.

Then open up a two browser windows and go to `localhost:4000` in each. Once you're there, open up the developer tools console in each window. For Chrome, they're under the View menu, View -> Developer -> Developer Tools. When the developer tools pane opens up, click on the "Console" link.

That should look like this.



With the server started, we already have a stateful backend running. The good news is that we won't have to build out a full frontend application to exercise this. Your favorite browser's JavaScript engine is a fine stateful frontend environment to work with. The browser's developer tools console will let us run code directly in that environment.

Here's the plan for how we'll work for the rest of the chapter. We'll write some code in a new channel module and recompile it in the IEx session. Then we'll go into the JavaScript console, push some messages down the channel to exercise that code, and then check in the IEx console to see what happened. We'll follow that pattern for each new piece of functionality we build.

Define a New Module

The first thing we need is a channel module. Let's get one started at /web/channels/game_channel.ex.

channel/lib/game_channel.ex

```
defmodule IslandsInterface.GameChannel do
  use IslandsInterface.Web, :channel

  alias IslandsEngine.Game
```

In order to make it behave like a channel, we use IslandsInterface.Web :channel. That triggers the channel/0 function in IslandsInterface.Web, located at /web/web.ex.

```
def channel() do
  quote do
    use Phoenix.Channel
```

```
    import IslandsInterface.Gettext
  end
end
```

With the proto-channel module in place, there are two pieces of housekeeping that we need to check in on: routing requests to our socket, and registering our channel with a socket.

The "route" to our socket appears in our application's endpoint at lib/inslands_interface/endpoint.ex. Phoenix generated this for us when it created our project.

```
defmodule IslandsInterface.Endpoint do
  use Phoenix.Endpoint, otp_app: :islands_interface

  socket "/socket", IslandsInterface.UserSocket
  . . .
```

The next step is to define a socket so we can establish a connection to the channel. Phoenix generated one for us already at web/channels/user_socket.ex. All we need to do is register our new game channel there.

```
defmodule IslandsInterface.UserSocket do
  use Phoenix.Socket

  channel "game:*", IslandsInterface.GameChannel
```

The meaning here is that we want any messages with a topic that begins with "game:" to go through the GameChannel.

There's one more interesting piece to look at in IslandsInterface.UserSocket, the connect/2 function. Anytime a client attempts to connect to the socket, the request will make it's way through the connect/2 function Phoenix defined for us.

```
def connect(params, socket) do
  {:ok, socket}
end
```

This is a great spot to do any authentication work, or to assign any values to the socket so that they will pass through the system into the channel. For our purposes, we'll just pass right through by returning {:ok, socket}. We'll see the client side of this in action later.

With that configuration housekeeping out of the way, we're ready to start doing more with our game channel. We'll start with letting clients subscribe to it.

Join a Channel

Before clients can do anything more meaningful in a channel, they need to join it on a topic-subtopic combination. To let users do that, we need to implement a join/3 function in the IslandsInterface.GameChannel module.

For now, we'll do the simplest thing we can and just let anybody join. We do that by just returning {:ok, socket}. We'll see how to get a little more picky about letting players join a little later on.

```
def join("game:" <> _player, _payload, socket) do
  {:ok, socket}
end
```

join/3 always takes a topic-subtopic string, some form of payload, and a socket struct. The return will either be {:ok, socket} or {:error, %{reason: "<whatever reason you like>"}}. This is slightly different from other return tuples we'll see in a minute, but the {:ok, socket} return is an echo of the {:ok, state} tuple we return from init/1 when starting a GenServer.

Now that we have a simple clause of the join/3 function, let's go to the IEx session we have running and compile our new GameChannel.

```
iex> c "web/channels/game_channel.ex"
[IslandsInterface.GameChannel]
```

Great, that's just what we wanted to see. Now we need to get a client to use join/3. For that, we'll need some JavaScript, and we'll write some in the next section.

Establish a Client Connection

Our goal in this section is to write client code that can invoke the join/3 function we now have on the server. There are a few steps we'll need to take to make that happen.

We'll need to define a client socket and use it to establish a connection to the socket on the server. Then we'll need to define a new channel object on the client, and use it to join the channel on the server.

Phoenix ships with a JavaScript file that knows all about working with sockets and channels called phoenix.js. It's indispensable for writing JavaScript client code for channels, and our first task is to make it available in the browser's console window.

Let's go to player1's JavaScript console, that's the browser window on the left, and require the phoenix.js file.

```
> var phoenix = require("phoenix")
undefined
```

Next we need to instantiate a new socket object so we can establish a connection from the client to the channel running on the server. As we do that, we need to pass it a path to the socket as well as any parameters we want to pass in as we establish a connection. We don't need to pass any in, so we use a blank object.

```
> var socket = new phoenix.Socket("/socket", {})
undefined
```

Once we have the socket object, we can have it establish a connection to the path we defined when we created it.

```
> socket.connect()
undefined
```

Now that we're connected, we need to define a new channel object before we're ready to start pushing messages down the server. To do that, we invoke the socket.channel function with the topic-subtopic, and some parameters.

The general form of that call looks like this.

```
> var new_channel = socket.channel("topic:subtopic", {some_key: "some_value"})
```

The parameters are important. The ones we specify here are the ones that will get passed to the join/3 function in the GameChannel, even if we pass other parameters into the client's join function later on.

```
def join("game:" <> _player, parameters, socket) do
```

To make this a little neater and more flexible, let's wrap that socket.channel call in a new function called new_channel. It will take a subtopic, and the screen name of the player who wants to join. The one parameter we want to send into the channel object itself is the player's screen name.

Go ahead and type the function definitions in at player1's console prompt.

```
> function new_channel(subtopic, screen_name) {
 return socket.channel("game:" + subtopic, {screen_name: screen_name});
}
undefined
```

Now we can invoke the new_channel function with a player's name to generate a new channel object. This will already have the parameters we specified baked into it.

```
> var game_channel = new_channel("moon", "moon")
undefined
```

If we click on the game_channel object in the console to inspect it, we'll see the params object itself.

```
> game_channel
Channel
. . .
params: Object
  screen_name: "moon"
  __proto__: Object
. . .
```

That's exactly what we want.

There's one more function we'll need to define in player1's console, and that's the join function itself.

As a practical matter, the game_channel object already has a join function. We could call it directly, but the return value would be an object that, on the face of it, wouldn't tell us whether it worked or not.

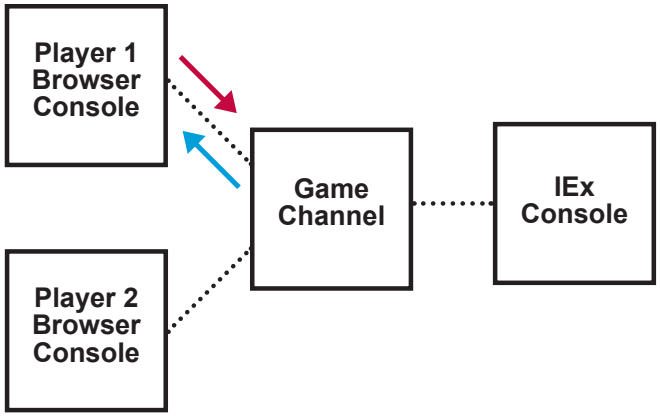
In order to see that something is really happening in the JavaScript console, we'll define a wrapper function around the channel's join function.

```
> function join(channel) {
  channel.join()
    .receive("ok", response => {
        console.log("Joined successfully!", response)
      })
    .receive("error", response => {
        console.log("Unable to join", response)
      })
}
undefined
```

Inside our wrapper function, we chain receive function calls to check for a return of "ok" or "error" and log out a different message depending on which one we get.

Now that we have our own join function defined, we can invoke it, and if all goes well, that will invoke the join/3 function we wrote in the channel.

This message path is going to be from player1 to the channel and only back to player1, like this.



Let's try it out in player1's console.

```
> join(game_channel)
undefined
Joined successfully! Object {}
```

That worked!

Back in our IEx session, we'll see that the join call worked.

```
iex> JOIN game:moon to IslandsInterface.GameChannel
  Transport:  Phoenix.Transports.WebSocket
  Parameters: %{"screen_name" => "moon"}
[info] Replied game:diva :ok
```

Clients can also leave a channel. We can give the same treatment to a leave function.

```
> function leave(channel) {
  channel.leave()
   .receive("ok", response => {
       console.log("Left successfully", response)
     })
   .receive("error", response => {
       console.log("Unable to leave", response)
     })
}
```

Now that we can get a client to join a channel on the server, we're ready to get a dialog going between the client and the server.

Converse Over a Channel

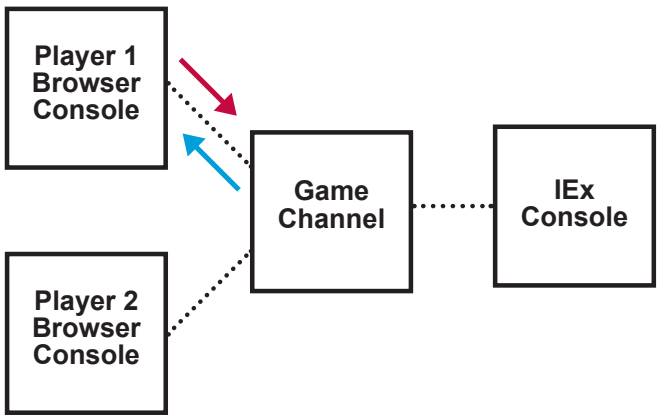
This is where things start to get interesting. So far, we've sent a join message and gotten an ok back, but Channels support much richer forms of bi-directional communications between clients and the server. We're going to take a closer look at the most common of these, and they will help us a lot through the rest of this chapter and while building your own applications.

The simplest way to send a message back from the server is to return a :reply tuple from the channel callback. This is a three element tuple {:reply, some_response, socket}, where the middle element gets sent back to the client that originally sent the message. This should look really familiar after our work with GenServer and :gen_statem.

Another way to talk back from a channel is with the push/3 function. push/3 sends the caller back a string that represents a new event along with a data payload. The caller needs to listen for that event, and we can define whatever actions to take in response to it that we want.

Broadcasting is another way for the server to send messages back to clients. The broadcast/3 and broadcast!/3 functions also send a string that represents a new event, along with a payload, but they send that event to every client that has subscribed to that specific topic-subtopic.

Let's take a look at reply tuples first. The path of the message passing will go from player1, to the channel on the server, and back to player1.



Just as we saw in OTP Behaviours, there's a mapping in channels between function calls and specific callbacks. When a channel object in the client calls

the push function with a message and a payload, that triggers the handle_in/3 callback in the channel on the server.

Also as in OTP Behaviours, we'll define multiple clauses of handle_in/3 that each pattern match on a different message, or a different payload.

The first thing we'll need in the GameChannel is a clause of handle_in/3 that matches the message "hello" and replies with :ok.

```
def handle_in("hello", payload, socket) do
  {:reply, {:ok, payload}, socket}
end
```

Remember to recompile the GameChannel module in the IEx session, otherwise this callback function won't exist in the running BEAM.

```
iex> r IslandsInterface.GameChannel
warning: redefining module IslandsInterface.GameChannel
(current version loaded from
_build/dev/lib/islands_interface/ebin/Elixir.IslandsInterface.GameChannel.beam)
web/channels/game_channel.ex:1

{:reloaded, IslandsInterface.GameChannel, [IslandsInterface.GameChannel]}
```

We could trigger this callback from player1's console with game_channel.push(), but we would get an object back that wouldn't tell us much on first glance.

To see how this behaves more clearly, let's write another wrapper function in player1's console that will give us some feedback. For the payload, we'll pass a JavaScript object with greeting.

```
> function say_hello(channel, greeting) {
  channel.push("hello", {"message": greeting})
    .receive("ok", response => {
        console.log("Hello", response.message)
      })
    .receive("error", response => {
        console.log("Unable to say hello to the channel.", response.message)
      })
}
undefined
```

Let's check out how this works in player1's console.

```
> say_hello(game_channel, "World!")
undefined
Hello World!
```

Of course, we can do anything we would like before we return the payload, as long as we adhere to the rules for the return value. We need to reply with

either a status atom—:ok or :error—by itself, or a tagged tuple with a status atom and a map for the second element.

Just to prove that the error condition works, let's change the function to alter the payload and always return an error.

```
def handle_in("hello", payload, socket) do
  payload = %{message: "We forced this error."}
  {:reply, {:error, payload}, socket}
end
```

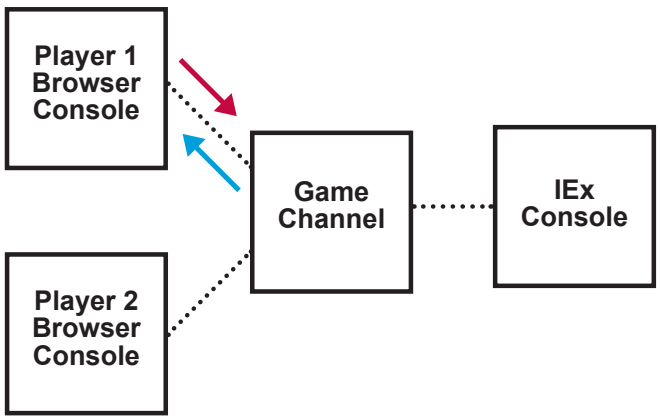
Go ahead and recompile the GameChannel in the IEx session.

Then let's go to player1's browser console and send a "hello" message to the channel again.

```
> say_hello(game_channel, "World!")
undefined
Unable to say hello to the channel. We forced this error.
```

Now let's take a look at how push/3 works inside the GameChannel. Instead of a reply tuple, this will send a new event down the channel, but only to the original caller. Since we're not relying on a reply tuple to communicate back to the client, we'll replace it with {:noreply, socket}.

The path of message passing will look the same as reply.



Let's go to the channel and change the handle_in/3 clause to use push/3. We could leave the :reply tuple there and get two responses, but since push/3 will already send a reply, it makes more sense to swap in {:noreply, socket} instead.

Let's recompile the GameChannel so we can see this in action.

```
def handle_in("hello", payload, socket) do
  push socket, "said_hello", payload
  {:noreply, socket}
end
```

Then let's head over to player1's console to give it a try.

```
> say_hello(game_channel, "World!")
undefined
```

And we get nothing.

Here's why. The handle_in/3 clause is pushing a new event, "said_hello" to the caller. The problem is that we don't have any code in the browser that is listening for that event.

We need to use the on function defined on the channel object to listen for the "said_hello" event and respond in a way we define. We'll just have it log "Returned Greeting" and the response to the console.

Let's go ahead and define an event listener in player1's console and try again.

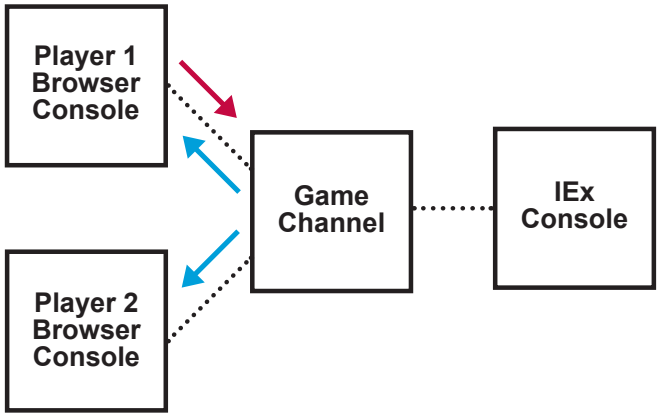
```
> game_channel.on("said_hello", response => {
    console.log("Returned Greeting:", response.message)
  })
undefined
```

```
> say_hello(game_channel, "World!")
undefined
Returned Greeting: World!}
```

That's exactly what we want.

Now let's take a look at, broadcast/3 or the bang version broadcast!/3. The bang version will raise an exception if it fails. The non-bang version will return an error tuple instead.

The message path for broadcast is going to look a little different. The message will originate from player1 to the channel, but the channel will send event messages to both player1 and player2.



This will also send an event, which is the second argument after the socket, to all members who have joined the channel.

Let's use the bang version to simplify our code. We can change our "hello" clause of handle_in/3 to look like this.

```
def handle_in("hello", payload, socket) do
  broadcast! socket, "said_hello", payload
  {:noreply, socket}
end
```

Since we're going to broadcast an event to all users who have joined the channel instead of returning a reply, use the {:noreply, socket} return value.

Now to see this really work, let's go over to Player2's console and get it set up to communicate over the channel. We'll need to do all the setup we did before—requiring the Phoenix.js file, defining a socket, connecting, and joining the channel.

```
> var phoenix = require("phoenix")
undefined

> var socket = new phoenix.Socket("/socket", {})
undefined

> socket.connect()
undefined

> function new_channel(player, screen_name) {
 return socket.channel("game:" + player, {screen_name: screen_name});
}
undefined

function join(channel) {
  channel.join()
```

```
    .receive("ok", response => {
        console.log("Joined successfully!", response)
      })
    .receive("error", response => {
        console.log("Unable to join", response)
      })
}
undefined
```

We want to be sure to communicate on exactly the same topic-subtopic, so we pass in the same name, "moon" to the join function that we did in Player1's console. But this is a different player, so we pass in a different screen name.

```
> var game_channel = new_channel("moon", "diva")
undefined

> join(game_channel)
undefined
Joined successfully Object {}
```

Now we need to make sure that Player2's console is listening for the "said_hello" event, just like Player1.

```
> game_channel.on("said_hello", response => {
    console.log("Returned Greeting", response.message)
  })
undefined
```

Now for the interesting part. Let's go back to Player1's console and say hello again.

```
> say_hello(game_channel, "World!")
undefined
Returned Greeting: World!
```

Great, it caught the event. If we take a look at player2's console, we'll see that it caught the same event.

That's exactly what we want to see.

Connect the Channel to the Game

Here's where we begin to do the real work of getting our new game channel talking to the game server. We're going to build out new clauses of handle_in/3 that correspond to the public interface functions we wrote for IslandsEngine.Game. We'll pick the right communication strategy for each action, and we'll continue to check our work at each step, both in the browser consoles and in IEx.

The mechanics of making these connections is so easy, it's going to feel like cheating. When we're done, all the actions a player can take in the game will be exposed on the web through a channel interface.

For all of these new actions, we're going to follow the same pattern we used for the "hello" function. We'll define a new clause of handle_in/3 that calls directly to the game server, define a function in the browser console that pushes a message to that clause, call the function, and check the response.

Now that we're talking to the game server, though, we have the opportunity to check the game server state before and after we call the function in the console to make sure that the action worked.

We're ready to go, and the place to begin is with initializing a new game.

Start a New Game

This is where we begin to expose the actual game to the web. We'll start with a clause of handle_in/3 that will match on the "new_game" action. Within that function, we'll call directly into the Game.start_link/1 function to start the game, and report back on the success or failure to start the GenServer.

All we need to start a new game is the first player's name. We decided in this chapter that the first player's name will also be the subtopic of the channel. Because of that, we don't need to pass in the player's name to start the game, we can derive it from the topic stored on the socket struct.

Since there will be only one player at this point, we only need to respond to that one player. Sending back a :reply tuple fits the bill.

```
def handle_in("new_game", _payload, socket) do
  "game:" <> player = socket.topic
  case Game.start_link(player) do
    {:ok, _pid} ->
      {:reply, :ok, socket}
    {:error, reason} ->
      {:reply, {:error, %{reason: reason}}, socket}
  end
end
```

Before we try this in the browser, let's make sure that the game server process hasn't yet been started.

Head over to the IEx session we have running and check for a game process named after the first player who joined the channel. We'll also need to recompile the GameChannel.

```
iex> GenServer.whereis({:global, "game:moon"})
```

```
nil
```

Great, there is no process with that global name.

Back in player1's browser console, let's define a new function that will push the "new_game" message to the channel.

```
> function new_game(channel) {
 channel.push("new_game")
   .receive("ok", response => {
       console.log("New Game!", response)
     })
   .receive("error", response => {
       console.log("Unable to start a new game.", response)
     })
}
undefined
```

And when we call that function in player1's console, it works!

```
> new_game(game_channel)
undefined
New Game! Object {}
```

Now let's check back in IEx to see if the game process exists.

```
iex> GenServer.whereis({:global, "game:moon"})
#PID<0.402.0>
```

Great! We started a new game from the browser.

Now let's see what happens when we do something we know will fail, like starting a game with the same name. Recall that we can only start one game with the same name at a time.

```
> new_game(game_channel)
undefined
Joined successfully Object {}
```

That's strange. The console says that we rejoined.

This happens because the channel crashed. After the crash, the client tried, and succeeded, to reconnect.

But why did the channel crash? Won't GenServer throw a polite error if we try to start another game with the same name? Yes, it will, but the polite error it throws includes a PID. Poison, the JSON encoder, doesn't know how to encode PIDs. Let's take a look back at the IEx console to see the evidence.

```
[error] GenServer #PID<0.375.0> terminating
** (Poison.EncodeError) unable to encode value:
            {:already_started, #PID<0.376.0>}
```

```
. . .
```

The easiest thing we can do to fix this is just pass the reason for the error through the inspect/1 function in the error reply. Then the handle_in/3 for new_game would look like this.

channel/lib/game_channel.ex

```
def handle_in("new_game", _payload, socket) do
  "game:" <> player = socket.topic
  case Game.start_link(player) do
    {:ok, _pid} ->
      {:reply, :ok, socket}
    {:error, reason} ->
      {:reply, {:error, %{reason: inspect(reason)}}, socket}
  end
end
```

After we recompile the game channel, and try to start the game twice, we get an entirely different result.

```
> new_game(game)
undefined
Unable to start a new game. Object {reason: "{:already_started, #PID<0.386.0>}"}
```

That's much nicer.

Now it's time to work on adding a second player.

Add a Second Player

In order to add a second player, we just need to pass that player's name to the Game.add_player/2 function along with the global name for the game.

Successfully adding a second player is something we want both players to know about. But if we fail to add the second player, only that player really needs to know. This is a case where we can use broadcast!/3 on success, and :reply tuple if something goes wrong.

There's a lovely coincidence here. The socket.topic, "game:moon" just happens to be the global name that we've given the game server, so we can use it to directly address that process.

With that information, let's define a new handle_in/3 clause for the "add_player" action.

channel/lib/game_channel.ex

```
def handle_in("add_player", player, socket) do
  case Game.add_player({:global, socket.topic}, player) do
    :ok ->
      broadcast! socket, "player_added", %{message:
```

```
      "New player just joined: " <> player}
      {:noreply, socket}
    {:error, reason} ->
      {:reply, {:error, %{reason: inspect(reasonn)}}, socket}
  end
end
```

Now let's go over to player2's browser console and add a function to push the "add_player" message to the channel, with the new player's name as the payload.

```
> function add_player(channel, player) {
  channel.push("add_player", player)
    .receive("error", response => {
        console.log("Unable to add new player: " + player, response)
      })
}
undefined
```

In order to catch the "player_added" event on the channel, in the case where we are successful, we need to add a new on function to the game channel.

Let's add this to both player's browser consoles.

```
> game_channel.on("player_added", response => {
    console.log("Player Added", response)
  })
undefined
```

Now we can go back to the IEx session to check the state of the game. Since we've got a new game started, player1 should already have the name "moon", but player2 should not yet have a name.

```
iex> game = IslandsEngine.Game.call_demo({:global, "game:moon"})
%IslandsEngine.Game{fsm: #PID<0.598.0>, player1: #PID<0.489.0>,
 player2: #PID<0.597.0>}

 iex> IO.puts IslandsEngine.Player.to_string(game.player1)
 %Player{:name => "moon",
   . . . }"

iex> IO.puts IslandsEngine.Player.to_string(game.player2)
%Player{:name => :none,
  . . . }"
```

That's exactly what we wanted.

In the second player's browser console, let's actually add the second player.

```
> add_player(game_channel, "diva")
undefined
Player Added Object {message: "New player just joined: diva"}
```

Great, we've captured the "player_added" event and logged its message to the console.

If we check in the first player's browser console, we should see the message there as well.

Finally, let's take a look at the game state in IEx to make sure that player2 has a name.

```
iex> IslandsEngine.Player.to_string(game.player1)
"%Player{:name => "moon",
  . . . }"
iex> IslandsEngine.Player.to_string(game.player2)
"%Player{:name => "diva",
  . . . }"
```

Yes, that worked exactly the way we expected it to.

Now let's move on to setting an island's coordinates.

Setting Island Coordinates

Setting island coordinates requires a player, an island, and the coordinates to set in the island. This is an action that needs to be secret, for the eyes of the player setting their island coordinate only. Giving that information away is like giving the game away.

As we define a new handle_in/3 clause to do this, we're going to use a :reply tuple, so that only the player setting their island's coordinates will see the response.

The "set_island_coordinates" message originates in JavaScript. JavaScript doesn't have an atom type, so we'll send these parameter values over as strings. That means we'll need to convert them to atoms in the handle_in/3 function before we pass them into the game server.

Atoms are really appropriate in the Elixir world, but they don't exist in JavaScript. We're ok doing these translations here because this is a boundary of the system. There won't be any other way to interact with the game engine from the web.

channel/lib/game_channel.ex

```
def handle_in("set_island_coordinates", payload, socket) do
  %{"player" => player, "island" => island, "coordinates" => coordinates} = payload
  player = String.to_atom(player)
  island = String.to_atom(island)
  coordinates = Enum.map(coordinates, fn coord -> String.to_atom(coord) end )
  case Game.set_island_coordinates({:global, socket.topic},
                                   player, island, coordinates) do
    :ok ->  {:reply, :ok, socket}
```

```
    :error -> {:reply, :error, socket}
  end
end
```

As we have with all these new actions so far, we'll need a new function to wrap the channel.push call and show us what the result is. Let's add this to both player's browser consoles.

```
> function set_island_coordinates(channel, player, island, coordinates) {
 params = {"player": player, "island": island, "coordinates": coordinates}
 channel.push("set_island_coordinates", params)
   .receive("ok", response => {
       console.log("New coordinates set!", response)
     })
   .receive("error", response => {
       console.log("Unable to set new coordinates.", response)
     })
}
undefined
```

Before we set any coordinates in any islands, let's take a look at player1's island set.

```
iex> IslandsEngine.Player.get_island_set(game.player1) |>
...> IslandsEngine.IslandSet.to_string() |>
...> IO.puts
%IslandSet{atoll => []
dot => []
l_shape => []
s_shape => []
square => []
}
:ok
```

Great, all of player1's islands are empty, as we would expect.

Now let's go to player1's console and add the "a1" coordinate to their "atoll" island.

```
> set_island_coordinates(game_channel, "player1", "atoll", ["a1"])
undefined
New coordinates set! Object {}
```

Success! Now let's go to the IEx console and see if it really worked.

```
iex> IslandsEngine.Player.get_island_set(game.player1) |>
...> IslandsEngine.IslandSet.to_string() |>
...> IO.puts
%IslandSet{atoll => [(in_island:atoll, guessed:false)]
dot => []
l_shape => []
s_shape => []
```

```
square => []
}
:ok
```

It did work. Player1's atoll now clearly has a coordinate in it.

Before we move on to setting islands, let's get the game state ready for guessing coordinates later on. We'll add the "a1" coordinate to all the rest of player1's islands, and we'll add an island to one of player2's islands.

If we waited to do this until after we set the players islands, we'd need to start the sequence from the beginning before both players could guess, and potentially win.

First, let's add the "a1" coordinate to the rest of player1's islands in player1's console.

```
> set_island_coordinates(game_channel, "player1", "dot", ["a1"])
undefined
New coordinates set! Object {}

> set_island_coordinates(game_channel, "player1", "l_shape", ["a1"])
undefined
New coordinates set! Object {}

> set_island_coordinates(game_channel, "player1", "s_shape", ["a1"])
undefined
New coordinates set! Object {}

> set_island_coordinates(game_channel, "player1", "square", ["a1"])
undefined
New coordinates set! Object {}
```

Then let's populate one of player2's islands from their console.

```
> set_island_coordinates(game_channel, "player2", "atoll", ["c10"])
undefined
New coordinates set! Object {}
```

With that, we're ready to move on to setting player's islands.

Setting Islands

Once players are done moving their islands around, resetting their coordinates, they need to mark their islands as set in place. All the channel needs to do is pass an atom representing the player down to Game.set_islands/2 and pattern match on the result to send the right response.

A player successfully setting their islands is something we want both players to know about, so we'll use broadcast!/3 to respond when Game.set_islands/2 succeeds. If it fails, we only want to let that player know, so we use a :reply tuple.

channel/lib/game_channel.ex
```
def handle_in("set_islands", player, socket) do
  player = String.to_atom(player)
  case Game.set_islands({:global, socket.topic}, player) do
    :ok ->
      broadcast! socket, "player_set_islands", %{player: player}
      {:noreply, socket}
    :error -> {:reply, :error, socket}
  end
end
```

We'll need to transform the player from a string into an atom again because that's what the game server expects.

Now let's add a function wrapping the channel.push call to the server. If the function succeeds, we'll get an event back from the broadcast, so we won't need to chain in a .receive("ok") clause. We'll need this in both players' consoles.

```
> function set_islands(channel, player) {
 channel.push("set_islands", player)
   .receive("error", response => {
       console.log("Unable to set islands for: " + player, response)
    })
}
undefined
```

Now we'll need to listen for the "player_set_islands" event the server will broadcast on success. We'll need this in both players' consoles as well.

```
> game_channel.on("player_set_islands", response => {
    console.log("Player Set Islands", response)
  })
undefined
```

There won't be any changes in the game server state to check here, but the :gen_statem state will change along the way. Let's check in with that in the IEx session. We would expect the state machine to be in the :players_set state at this point, and we would expect it to stay there until both players have set their islands.

```
iex> alias IslandsEngine.Rules
IslandsEngine.Rules

iex> # We should already have the game variable bound to the PID.

iex> fsm = Game.call_demo(game).fsm
#PID<0.605.0>

iex> Rules.show_current_state(fsm)
:players_set
```

That's exactly what we expected.

Now let's call the set_islands function in player1's browser window.

```
> set_islands(game_channel, "player1")
undefined
Player Set Islands Object {player: "player1"}
```

Great. We got the "player_set_islands" event signifying success. We should also see the same response in player2's console.

Now let's check back with the state machine in the the IEx session. It should still be in the :plays_set state.

```
iex> Rules.show_current_state(fsm)
:players_set
```

Nice, that's just what we expected to see.

Let's set player2's islands in their console. This should transition the state machine to the :player1_turn state.

```
> set_islands(game_channel, "player2")
undefined
Player Set Islands Object {player: "player2"}
```

The call was successful, and we should see Player Set Islands Object {player: "player2"} in both consoles.

In the IEx session, we should see that it's player1's turn.

```
iex> Rules.show_current_state(fsm)
:player1_turn
```

That's beautiful—exactly what we wanted.

Now that we can set player's islands, we're off to guessing coordinates, the last and arguably most important part of playing the game.

Guessing Coordinates

For this final piece, we'll need to pass a player and a coordinate into Game.guess_coordinate/3. We should show the result of a successful guess to both players, so we'll broadcast those. If a guess fails, we'll return a :reply tuple.

That's exactly what we've done for the past few handle_in/3 clauses, but this one has a twist. The response we'll get back from Game.guess_coordinate/3 will be a tuple, but we need to use a map for the payload of our broadcast. That means that we'll need to do a little pattern matching to destructure the tuple and build it back up again as a map.

channel/lib/game_channel.ex

```
def handle_in("guess_coordinate", params, socket) do
```

```
  %{"player" => player, "coordinate" => coordinate} = params
  player = String.to_atom(player)
  coordinate = String.to_atom(coordinate)
  case Game.guess_coordinate({:global, socket.topic}, player, coordinate) do
    {:hit, island, win} ->
      result = %{hit: true, island: island, win: win}
      broadcast! socket, "player_guessed_coordinate", %{player: player, result: result}
      {:noreply, socket}
    {:miss, island, win} ->
      result = %{hit: false, island: island, win: win}
      broadcast! socket, "player_guessed_coordinate", %{player: player, result: result}
      {:noreply, socket}
    {:error, reason} -> {:reply, {:error, %{player: player, reason: reason}}, socket}
  end
end
```

Great, now let's add a wrapper function to push this message to the channel. We'll need it in both players' consoles.

```
> function guess_coordinate(channel, player, coordinate) {
  params = {"player": player, "coordinate": coordinate}
  channel.push("guess_coordinate", params)
    .receive("error", response => {
        console.log("Unable to guess a coordinate: " + player, response)
      })
}
undefined
```

Since we're broadcasting from the handle_in/3 on success, we need to listen for the "player_guessed_coordinate" event in both player's consoles.

```
> game_channel.on("player_guessed_coordinate", response => {
    console.log("Player Guessed Coordinate: ", response.result)
  })
undefined
```

It's always player1's turn first, but before they guess a coordinate, let's take a look at player2's islands.

```
iex> IslandsEngine.Player.get_island_set(game.player2) |>
...> IslandsEngine.IslandSet.to_string() |>
...> IO.puts
%IslandSet{atoll => [(in_island:atoll, guessed:false)]
dot => []
l_shape => []
s_shape => []
square => []
}
:ok
```

That's great, it's got the coordinate we set in player2's atoll, and it hasn't yet been guessed.

Now let's let player1 make an incorrect guess.

```
> guess_coordinate(game_channel, "player1", "b10")
undefined
Player Guessed Coordinate:  player1
       Object {win: "no_win", island: "none", hit: false}
```

That tells us all the right things. It wasn't a hit, no island was forested, and it didn't result in a win.

Let's check player2's islands in IEx one more time to make sure they still look right.

```
iex> IslandsEngine.Player.get_island_set(game.player2) |>
...> IslandsEngine.IslandSet.to_string() |>
...> IO.puts
%IslandSet{atoll => [(in_island:atoll, guessed:false)]
dot => []
l_shape => []
s_shape => []
square => []
}
:ok
```

That's correct. They look exactly as they did before.

Now it's player2's turn to guess. There should be a single coordinate in each of player1's islands. Let's take a quick look in IEx to make sure.

```
iex> IslandsEngine.Player.get_island_set(game.player1) |>
...> IslandsEngine.IslandSet.to_string() |>
...> IO.puts
%IslandSet{atoll => [(in_island:square, guessed:false)]
dot => [(in_island:square, guessed:false)]
l_shape => [(in_island:square, guessed:false)]
s_shape => [(in_island:square, guessed:false)]
square => [(in_island:square, guessed:false)]
}
:ok
```

That's perfect, just what we want.

Recall that we set all of player1's islands to have a single `"a1"` coordinate. If we have player2 guess it, they should win.

```
> guess_coordinate(game_channel, "player2", "a1")
undefined
Player Guessed Coordinate:  player2
       Object {win: "win", island: "square", hit: true}
```

Player2 does in fact win. Just to make sure, let's look at player1's islands in IEx once again.

```
iex> IslandsEngine.Player.get_island_set(game.player1) |>
...> IslandsEngine.IslandSet.to_string() |>
...> IO.puts
%IslandSet{atoll => [(in_island:square, guessed:true)]
dot => [(in_island:square, guessed:true)]
l_shape => [(in_island:square, guessed:true)]
s_shape => [(in_island:square, guessed:true)]
square => [(in_island:square, guessed:true)]
}
:ok
```

Great, that's exactly what we should have seen.

That wraps up all the functionality the game itself needs. We could call the channel done if we wanted to and people could still play the game.

There's one nagging little bit though. We said early on that each game should be private to two players. We currently don't have a way to limit the number of players that can join the channel on a given topic-subtopic combination.

Phoenix Presence is going to help us out with this, and that's where we're going next.

Phoenix Presence

From the earliest days of Phoenix channels, developers have asked how to tell who is currently subscribed to a channel. Until recently, the answer was to create a custom solution that best fits an individual application's circumstances. But now, we have Phoenix Presence to solve that problem in a general way for all of us.

Presence has one job to do, to keep track of the clients subscribed to a topic on a channel. For us, that means keeping track of the players in each game. Presence does this amazingly well. This might sound like a trivial task, but it's deceptively difficult.

If you were to roll your own version of Presence, your first thought might be to maintain a list of subscribers, adding clients to the list when they join and removing them when they leave. This might work for a system with a single node.

With a single data structure on multiple nodes, though, you would have to make sure that the data is available to all nodes in the cluster. But nodes don't stay up forever, and a crash could lose all of the subscription data.

You could put the data in an external database to solve that data durability problem, but then network hiccups could disrupt communication to the database, and the system would miss clients joining or leaving. That would make the data out of sync with the real state of the channel.

The scenarios only get more complex from there. Add in users subscribing to the same topic from multiple devices as well as flakey mobile connections, and a solid solution might seem evasive indeed.

That's where CRDTs (conflict-free replicated datatypes) come to the rescue. They track the sequence of clients joining and leaving, across nodes and clients. They do it without relying on clock time, which can drift from machine to machine. CRDTs allow Presence to reconstruct the sequence of events to accurately determine which clients are currently subscribed to a topic on a channel.

In the future, this job might even be expanded to keep track of other things that can join or leave a group, like nodes, services, and processes. Stay tuned!

The power of Phoenix Presence is compounded by how easy it is to setup and use. We'll need a new and mostly empty Presence module, and we'll need to make sure that it's started when IslandsInterface is.

After that, we'll need a single callback function in the channel to make it work, though we'll add another just to help us see Presence in action.

The plan is that as a player is joining, before we return `{:ok, socket}`, the channel will send itself a message with the player's screen name. The first callback we write will match that message and have Presence start tracking that screen name.

The browser consoles we've been using are already full of state from playing a game. Let's start with a clean slate by shutting the server down and reloading both players' windows.

We need to begin with an empty presence module in our application. Let's put it in `/lib/islands_interface/presence.ex`.

```
defmodule IslandsInterface.Presence do
  use Phoenix.Presence, otp_app: :islands_interface,
                        pubsub_server: IslandsInterface.PubSub
end
```

This module brings in `Phoenix.Presence`, specifies our OTP application name, and specifies which pubsub server we'll use. We're using the one that Phoenix provided for us when we generated the project.

We won't need to add anything to this module. It's fine as it is.

Next we need to make sure that the presence module gets started in our supervision tree when we start the application. Open up /lib/islands_engine.ex and add the presence module in the list of children.

```
children = [
  supervisor(IslandsInterface.Endpoint, []),
  supervisor(IslandsInterface.Presence, []),
]
```

That's it for the application setup. Hardly anything to it.

Before we can begin to use Presence, in the GameChannel, we'll need to alias it.

```
alias IslandsInterface.Presence
```

Now we need a single callback in game_channel.ex. In order to handle a raw message the channel will send to itself, we'll use a handle_info/2 callback. We'll have this callback match on {:after_join, player_name}.

channel/lib/game_channel.ex

```
def handle_info({:after_join, screen_name}, socket) do
  {:ok, _} = Presence.track(socket, screen_name, %{
    online_at: inspect(System.system_time(:seconds))
  })
  {:noreply, socket}
end
```

The body of the callback tells Presence to start tracking this user by their screen name, and notes when they joined the channel.

The way we trigger that callback is by having the channel send itself the {:after_join, screen_name} message we matched for in the handle_info/2 callback.

```
def join("game:" <> _player, %{"screen_name" => screen_name}, socket) do
  send(self(), {:after_join, screen_name})
  {:ok, socket}
end
```

In order to see presence info from the browser, though, we'll need another handle_in clause that will broadcast the list of players that Presence is currently tracking.

channel/lib/game_channel.ex

```
def handle_in("show_subscribers", _payload, socket) do
  broadcast! socket, "subscribers", Presence.list(socket)
  {:noreply, socket}
end
```

Then we'll need to listen for and respond to the "subscribers" event in both player's consoles.

```
> game_channel.on("subscribers", response => {
    console.log("These players have joined: ", response)
  })
```

Let's go ahead and test this out. In each player's console, let's begin to set up the state, just as we did before.

```
> var phoenix = require("phoenix")
undefined

> var socket = new phoenix.Socket("/socket", {})
undefined

> socket.connect()
undefined

> function new_channel(player, screen_name) {
 return socket.channel("game:" + player, {screen_name: screen_name});
}
undefined

> function join(channel) {
  channel.join()
    .receive("ok", response => { console.log("Joined successfully!", response) })
    .receive("error", response => { console.log("Unable to join", response) })
}
undefined
```

Then in player1's console, let's create a new channel object and listen for the "subscribers" event.

```
> var game_channel = new_channel("moon", "moon")
undefined

> game_channel.on("subscribers", response => {
    console.log("These players have joined: ", response)
  })
undefined
```

Now let's do the same thing for player2 in their console.

```
> var game_channel = new_channel("moon", "diva")
undefined

> game_channel.on("subscribers", response => {
    console.log("These players have joined: ", response)
  })
undefined
```

With both players set up, let's have player1 join the channel. After player1 joins, we can push the "show_subscribers" message over the channel.

```
> join(game_channel)
undefined
Joined successfully! Object {}

> game_channel.push("show_subscribers")
. . .
These players have joined:  Object {moon: Object}
```

That correctly tells us that the first player has joined. Nice.

Even though we're broadcasting from the channel, we won't yet see the logged message in player2's console because they haven't joined the channel yet.

Let's take care of that now.

```
> join(game_channel)
undefined
Joined successfully! Object {}

> game_channel.push("show_subscribers")
. . .
These players have joined:  Object {moon: Object, diva: Object}
```

That's perfect. In player2's console, we see a message telling us that both players have joined. Since player1 had joined previously, they'll see this logged message as well.

Now that we can tell which players are subscribed to a channel on a given topic, we can implement the one last feature we need, authorization.

Authorization

In this last section, we'll be tackling authorization—deciding if an action is permissible. The action we really care about is joining a channel. The rules for this are simple. We only want two players to join a channel on any given topic-subtopic, and we want those two players to have different screen names.

Authentication Resources

In Islands, we don't have any need for authentication—determining if users are who they say they are. If your application does need authentication, there are resources out there to help. Check out the documentation for Phoenix.Token if you need token based authentication. Alternately, Programming Phoenix by Chris McCord, José Valim, and Bruce Tate has a lot of great information on authentication.

Now that we have presence, we can write functions to check both of the authorization conditions we outlined, and we can roll them up into a single function that determines whether a given player can join.

The first condition we need to check is how many players have already joined the channel. Presence.list/1 returns a map. The keys of this map are the screen names of all the players that have joined the channel on a specific topic-subtopic. We can write a function to return that number by getting the Presence list and counting the keys.

channel/lib/game_channel.ex
```
defp number_of_players(socket) do
  socket
  |> Presence.list()
  |> Map.keys()
  |> length()
end
```

We can also tell if a player is already subscribed to this game channel by seeing if a given screen name is already a key in the Presence map.

channel/lib/game_channel.ex
```
defp existing_player?(socket, screen_name) do
  socket
  |> Presence.list()
  |> Map.has_key?(screen_name)
end
```

With those two functions, we have enough information to see if a player is authorized to join the channel.

channel/lib/game_channel.ex
```
defp authorized?(socket, screen_name) do
  if number_of_players(socket) < 2 && !existing_player?(socket, screen_name) do
    true
  else
    false
  end
end
```

Now we can use this authorized?/2 function in join/3 to decide if we should let a new player join or not.

channel/lib/game_channel.ex
```
def join("game:" <> _player, %{"screen_name" => screen_name}, socket) do
  if authorized?(socket, screen_name)  do
    send(self(), {:after_join, screen_name})
    {:ok, socket}
  else
    {:error, %{reason: "unauthorized"}}
  end
end
```

To see this in action, let's stop the server and reload the two browser windows again, just to start from a clean slate. Then let's open a third window representing a third player that will not be able to join.

In each of the three browser consoles, go ahead and set up the state and functions that we'll need.

```
> var phoenix = require("phoenix")
undefined

> var socket = new phoenix.Socket("/socket", {})
undefined

> socket.connect()
undefined

> function new_channel(player, screen_name) {
 return socket.channel("game:" + player, {screen_name: screen_name});
}
undefined

> function join(channel) {
  channel.join()
    .receive("ok", response => { console.log("Joined successfully!", response) })
    .receive("error", response => { console.log("Unable to join", response) })
}
undefined
```

Now let's have each player instantiate a new channel object and try to join the channel, starting with player1. What we're expecting is that player1 and player2 will be able to join, but the third player won't.

```
> var game_channel = new_channel("moon", "moon")
undefined

> join(game_channel)
undefined
Joined successfully! Object {}
```

Player1 joined successfully.

Now let's try player2.

```
> var game_channel = new_channel("moon", "diva")
undefined

> join(game_channel)
undefined
Joined successfully! Object {}
```

Player2 joined without a problem, so let's try the third player in the new browser console.

```
> var game_channel = new_channel("moon", "nope")
```

```
undefined
> join(game_channel)
undefined
Unable to join Object {reason: "unauthorized"}
```

Great! As we expected, the third player wasn't allowed to join.

That's all the behavior we'll need from our channel.

This brings us right up to the boundary of conventional frontend web development. The steps that remain are to model the game state, render the player and opponent boards for each player, and map DOM events to the functions we just wrote, updating the game state appropriately along the way. There's nothing revolutionary from here on out, and there are a number of sources available show you how to do it in the frontend language and framework of your choice.

We won't leave you hanging though. We've created a GitHub organization to house frontend solutions for a variety of different frameworks and languages. I'll prime the pump with a React version and an Elm version, and I invite all of you to bring in your own solutions.

NOTE: *the GitHub organization is not yet complete.*

Wrapping Up

Congratulations! We've made it.

We've built a Phoenix channel that provides an interface to the game engine we wrote in Part 1. We did it without coupling the logic to the interface in any way. The game engine concerns itself entirely with the business logic of the game. The channel concerns itself entirely with brokering messages between clients and the game engine.

We've seen how to use Phoenix Presence to track clients who have subscribed to a channel on a topic, and we've seen how to use that to implement an authorization scheme for joining the channel.

With that, we've completed the last layer we're going to cover in this book. Now go build some amazing new web apps!

APPENDIX 1

Testing

Content to be supplied later.

APPENDIX 2

Installing System Dependencies

Content to be supplied later.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/lhelph
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/lhelph*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764