Extracted from:

Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenState, Flow, and Broadway

The Pragmatic Bookshelf

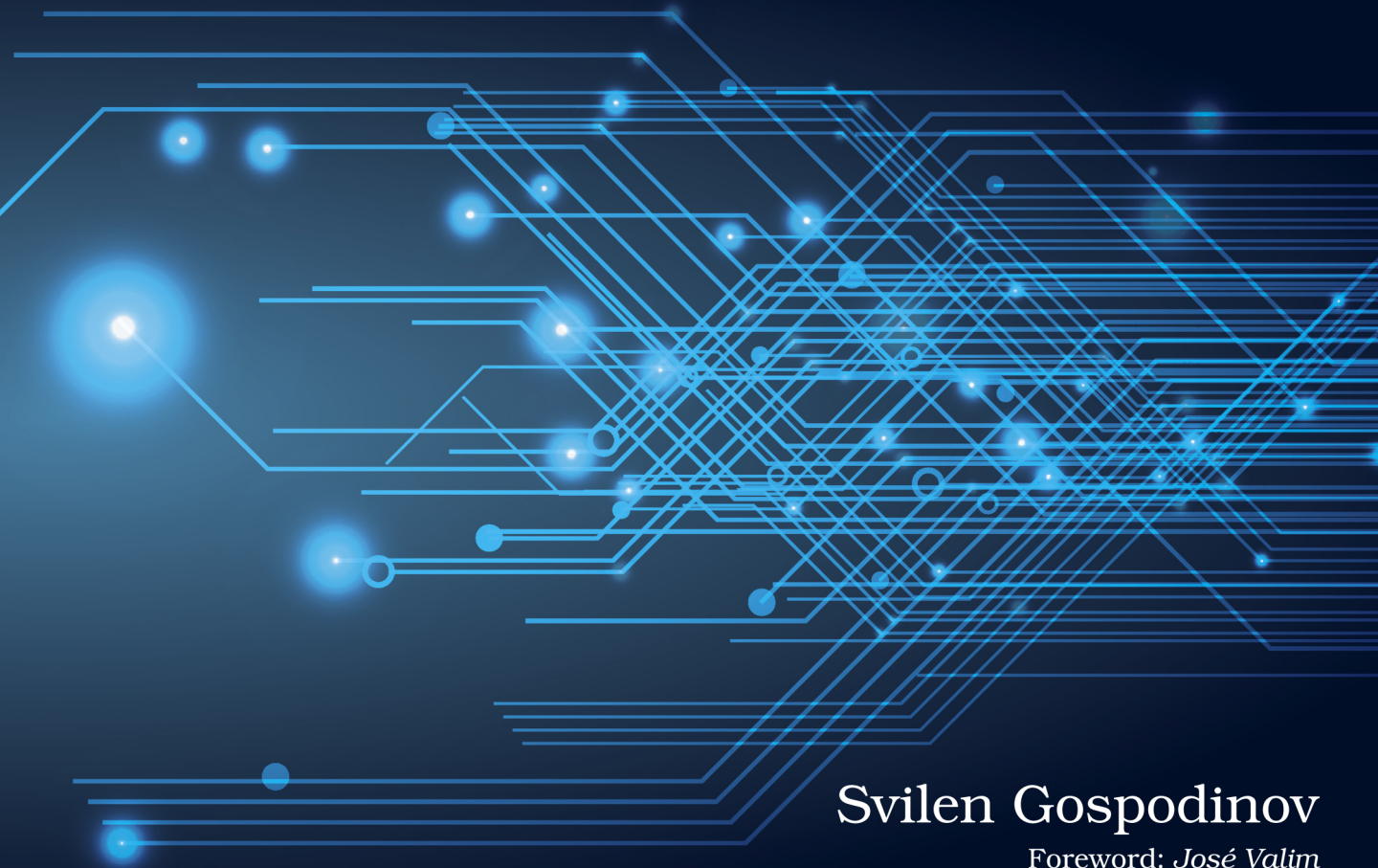
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenStage, Flow, and Broadway



Svilen Gospodinov

Foreword: *José Valim*
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

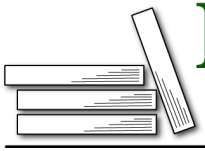
Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenState, Flow, and Broadway

Svilen Gospodinov

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

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

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

ISBN-13: 978-1-68050-819-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2021

CHAPTER 3

Data-Processing Pipelines with GenStage

In previous chapters, we covered several approaches to executing data asynchronously. Although different in their own way, they had one thing in common: we were deciding the amount of work that had to be done and Elixir would then eagerly process the work to give us the result.

This has some potential drawbacks. For example, we have a finite amount of memory and CPU power available. This means that our server may become overwhelmed by the amount of work it needs to do and become slow or unresponsive. Often we rely on third-party API services, which have rate limiting in place and fixed quotas for the number of requests we can make. If we go over their quota, requests will be blocked and our application will stop working as expected.

In a nutshell, as the amount of work we need to do grows, there is an increasing chance that we can hit a certain limit on a resource available to us. You can spend more money on computer hardware or buy more powerful virtual machines, but that's usually a temporary solution that often leads to diminishing returns. For that reason, making the best use of existing resources should be a priority when designing and scaling software applications.

In this chapter, we're going to learn how to build data-processing pipelines that can utilize our system resources reliably and effectively. We'll introduce the `GenStage` Elixir library and the fundamental building blocks that it provides. First we are going to create a simple data-processing pipeline to start with, then scale it and extend it to demonstrate how you can tackle more complex use cases.

But before we get into it, first we need to explain what *back-pressure* is, and how it enables us to build data-processing pipelines. Let's get started!

Understanding Back-Pressure

Imagine you're a famous writer giving autographs at a book event. There is a crowd of people rushing to meet you. You can only do one autograph at a time, so the organizers let people in one by one. When you sign someone's book, you ask for the next person to come forward. What if the organizers suddenly let everyone in? Of course, it will be complete chaos! You'll try to sign everyone's books as quickly as you can, but soon you'll get stressed and exhausted, leaving the event early.

It is much more efficient to have an orderly queue of people and to take your time to sign each book. Because you always ask for the next person to come forward, you are in control of the amount of work you have to do, and it is much easier to keep going. Maybe you won't get that tired, so you decide to stay at the event longer and make sure everyone gets an autograph. This is in fact an example of handling back-pressure in real life.

How does this translate in programming? Using the `GenStage` library we are going to build a data-processing pipeline that works like the well-organized book event we just described. The system will process only the amount of work it can handle at a given time, just like the famous writer from our example. If the system has free capacity, it will politely ask for more work and wait for it.

This simple shift in our thinking is very powerful. It enables us to build complex data pipelines that regulate themselves to utilize the available resources in the best possible way.

Borrowing Terminology

The term *back-pressure*, according to Wikipedia, originates from fluid dynamics and the automotive industry, where it is used to describe resistance to the normal flow of fluids in pipes.

Software engineers borrowed the term and loosely use it in the context of data processing, when something is slowing down or stopping the flow of data. When we talk about using back-pressure, we actually mean using a mechanism to control or handle back-pressure in a beneficial way.

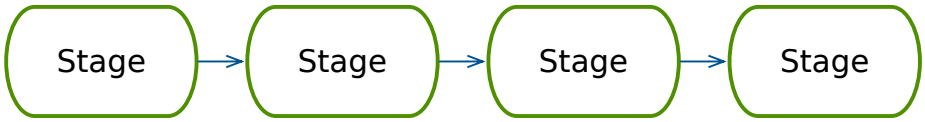
Introducing GenStage

`GenStage` was originally developed by José Valim, the creator of Elixir, and released in July 2016. As he described it in the official announcement:

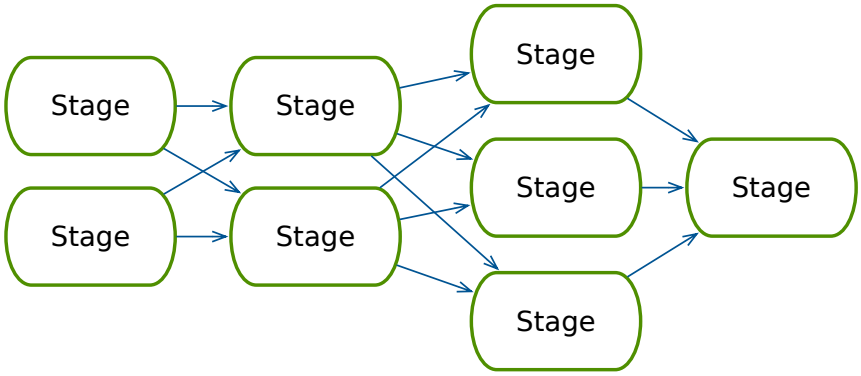
> "GenStage is a new Elixir behaviour for exchanging events with back-pressure between Elixir processes."

In the previous chapter, we used the GenServer behaviour to build long-running *server* processes. The GenStage behaviour, as its name suggests, is used to build *stages*. Stages are also Elixir processes and they're our building blocks for creating data-processing pipelines.

Stages are simple but very powerful. They can receive *events* and use them to do some useful work. They can also send events to the next stage in the pipeline. You can do that by connecting stages to each other, creating something like a chain, as you can see here:



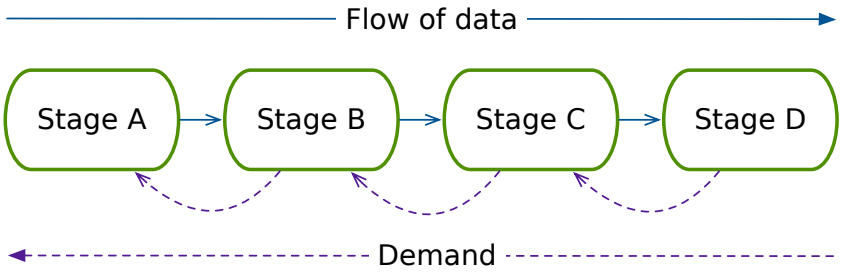
A stage can also have multiple instances of itself. Since stages are processes, this effectively means running more than one process of the same stage type. This means that you can also create something that looks like this:



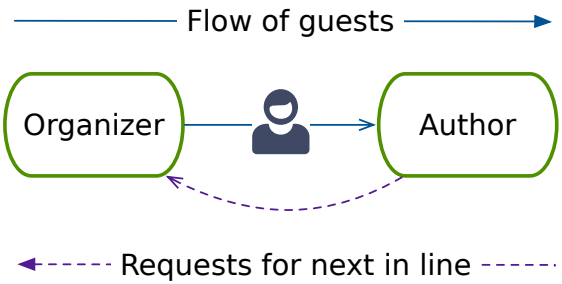
These are just two possible designs. In practice, different problems require different solutions, so your data-processing pipeline could end up looking completely different. When you finish this chapter, you will have a solid understanding of GenStage and you will be able to come up with a solution that works best for you.

As you can see, stages are very flexible and can be used in a variety of ways. However, their most important feature is back-pressure.

Although events move between stages from left to right on our diagram, it is actually the last stage in the pipeline that controls the flow. This is because the *demand* for more events travels in the opposite direction—from right to left. This figure shows how it works:



Stage D has to request events from Stage C, and so on, until the demand reaches the beginning of the pipeline. As a rule, a stage will send demand for events only when it has the capacity to receive more. When a stage gets too busy, demand will be delayed until the stage is free, slowing down the flow of data. Sounds familiar? This is exactly how the book event example from Understanding Back-Pressure, on page 6 would work if we implemented it as a data-processing pipeline. This figure illustrates how we can model the book event using stages:

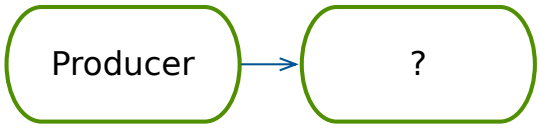


By treating guests as a flow of events, the author stage can "process" a guest by signing their book, and then request the next guest from the organizer when ready. The organizer stage, on the other hand, has to make sure that only one guest at a time comes forward, as the author requests. This is how back-pressure works in `GenStage`, and the best part is that you benefit from it simply by thinking in stages and connecting them together.

Connecting stages is easy, but first you need to know what stages to use. There are three different types of stages available to us: `producer`, `consumer`, and `producer-consumer`. Each one plays a certain role. Let's briefly cover each type of stage and see how it is made to work with the rest.

The Producer

At the beginning of a data-processing pipeline there is always a producer stage, since the producer is the source of data that flows into the pipeline. It is responsible for producing *events* for all other stages that follow. An *event* is simply something that you wish to use later on; it could be a map or a struct. You can use any valid Elixir data type. Here is an example of a two-stage data pipeline:



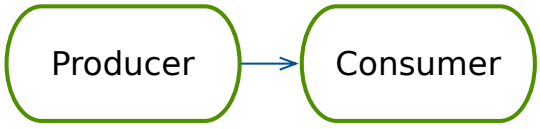
The events produced by the producer have to be processed by another stage, which is missing in the figure. What we need here is a consumer for those events.

The Consumer

Events created by the producer are received by the consumer stage. A consumer has to *subscribe* to a producer to let them know they're available, and request events.

This producer and consumer relationship is around us every day. For example, when you go to the farmer's market, the farmers that grow vegetables are producers. They wait until a customer (a consumer) comes and asks to buy some vegetables.

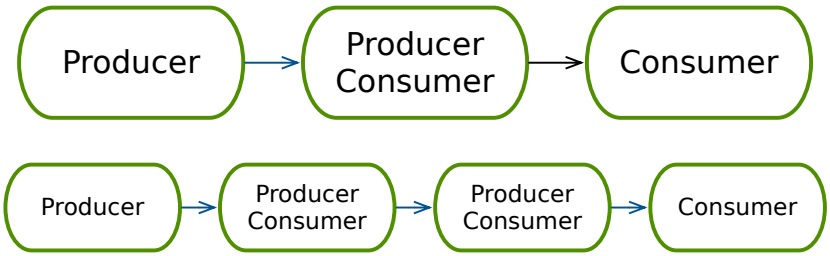
Consumer stages are always found at the end of the pipeline. Now we can fill in the missing process. The following figure illustrates the simplest data pipeline we can create using `GenStage`:



It has just a single producer and a consumer.

The Producer-Consumer

Although having a producer and a consumer is already very useful, sometimes we need to have more than two stages in our pipeline. This is where the producer-consumer stage comes in—it has the special ability to produce and consume events at the same time. The following figure shows how one or more producer-consumer stages can be used to extend a pipeline.



A useful analogy to a producer-consumer is the restaurant. A restaurant serves meals (producer) but in order to cook the meals, it needs ingredients sourced from its suppliers (acting as a consumer). In a nutshell, producer-consumers are the middle-man in our data-processing pipelines.

We've covered a lot of theory, so let's finally dive into some examples.