Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Design and documentation environment for Phoenix project generation templates

Estudante: Beltrán José Aceves Gil

Dirección: Laura Milagros Castro Souto

A Coruña, June of 2024.

To my loved ones

Acknowledgements

I would like to thank my family and friends for supporting me every step of the way and making this possible. And, of course, to my project director for sticking with me for such a long time, even in the face of many setbacks.

Abstract

The Phoenix framework, much like the broader Elixir ecosystem, strongly emphasises developer experience and ergonomics, with most of its features being well documented and easy to use. One of the notable exceptions is its code generation commands, which present serious usability concerns, discouraging the community from fully investing in such technology. This project aims to correct these issues by providing an environment to facilitate their use and documentation.

Resumo

El framework Phoenix, así como el ecosistema Elixir en general, pone un fuerte énfasis la experiencia y ergonomía de desarrollo, siendo la mayoría de sus características bien documentadas y fáciles de usar. Una de las notables excepciones son sus comandos de generación de código, que presentan serios problemas de usabilidad, lo cual dificultad la adopción de esta tecnologían en la comunidad. Este proyecto pretende proporcionar un entorno que facilite su uso y documentación.

Keywords:

- Elixir
- Phoenix Framework
- Developer Experience
- Ergonomics
- Livebook
- Code generation
- Documentation
- Design
- Development environment

Palabras chave:

- Elixir
- Framework Phoenix
- Experiencia de desarrollador
- Ergonomía
- Livebook
- Generación de código
- Documentación
- Diseño
- Entorno de desarrollo

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this first chapter, we will explain the context of this work, the motivation for the project, the main and secondary goals and the state of the art.

1.1 Context

Up until recently [1], developers could write a piece of software and expect it to run twice as fast two years down the line [2], commonly referred to as Moore's law [3]. This allowed CPU manufacturers to focus on single-core system development and, consequently, most programming languages have strong single-core performance. As improvements plateaued [4] and the industry transitioned into multi-core architectures , José Valim noticed that current mainstream programming languages [5] *"are not as effective as they could be"* [2] when dealing with this new paradigm.

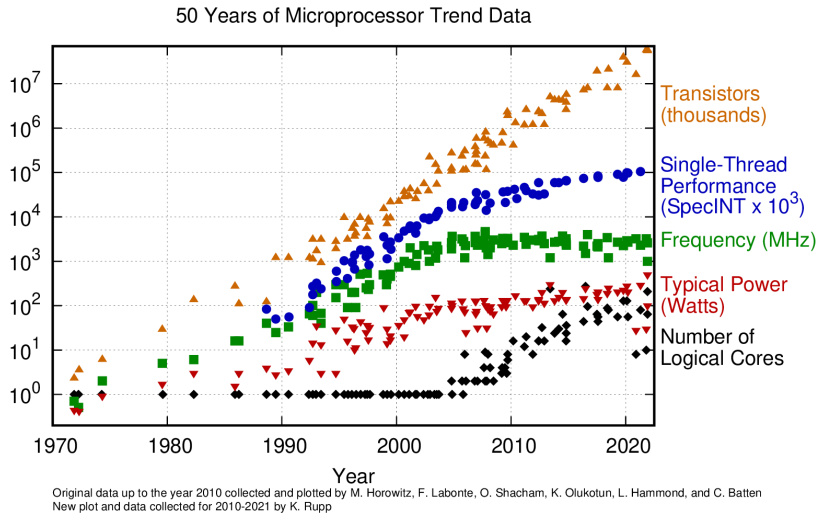

Figure 1.1: Evolution of Moore's Law over the years

He also realised that an existing technology, the BEAM/ErlangVM, was well-equipped to deal with challenges of concurrency, fault tolerance, and high availability[6]. He decided to build Elixir in order to bring into the spotlight the power of Erlang, a general-purpose, concurrent, functional high-level programming language [7] developed by Joe Armstrong et al.[8] in 1986 at Ericsson as a telecommunications programming language. Valim incorporated the ideas of rapid development and developer productivity of Ruby[9], of which he was a prominent contributor. This core idea of enabling developers to easily build concurrent and resilient systems permeated the Elixir community and its ecosystem, which would eventually grow and spawn a series of strong projects and tools. Two of the most notable are:

- **Livebook:** is a web application for writing interactive and collaborative code notebooks. It features markdown support, a rich code editor, diagram rendering, interactive widgets, remote code execution, and debugging of existing Elixir applications, one-click notebook deployments, and an easy-to-use version control format based on enhanced markdown [10]. It is commonly used as a collaborative prototyping environment [11], interactive documentation [12], or internal automation tool thanks to its live editors, easy deployment as applications [13], and the ability to combine text, code, diagram, and image blocks.

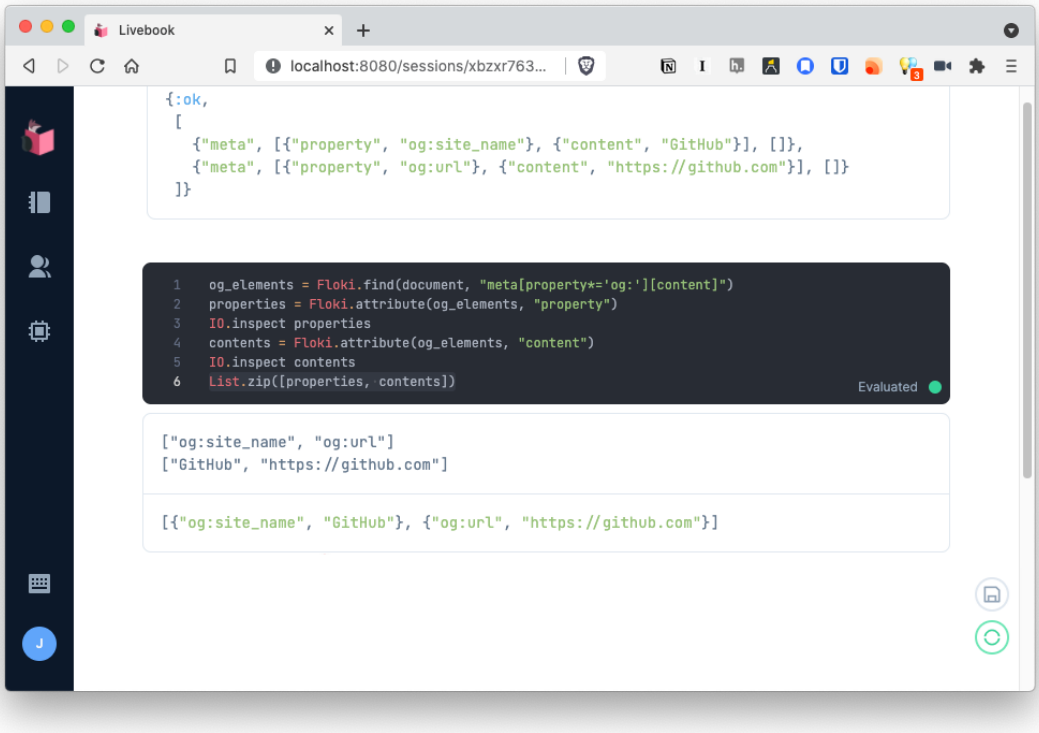

Figure 1.2: Livebook code execution cells

- **Phoenix Framework:** is a powerful web framework that uses the MVC pattern [14], boasting high developer productivity and application performance. Provides a comprehensive suite of built-in features, such as authentication, logging, system dashboards, real-time channels, PubSub, user presence, email notifications, database integration, testing, telemetry, and an innovative system to build interactive web applications with minimal JavaScript [15]. All of this while handling up to 2 million simultaneous user connections [16]. Phoenix has been voted the *most loved and admired framework* since 2022 in the **Stack Overflow Survey** [17, 18].

1.2 Motivation

Phoenix is a mature project released in 2015 [19], with a uniform level of polish of its features. Most of them are exploited through code, so they enjoy IDE features such as syntax highlighting, autocomplete, linter warnings and error messages, type annotations through Dialyzer [20], in-editor documentation, and AI code assistant integration. In contrast, its code generation capabilities, phx.gen [21], are accessed through a collection of CLI commands called Mix Tasks [22]. Thus, it does not enjoy the benefits of existing tooling and relies on classic terminal use conventions [23] where these quality-of-life enhancements are not typically available. This presents the following usability concerns:

- **Error prone input:** without syntax checker or linter [24], there is no feedback given to the user and any error will not be made apparent until the command is executed. Only then is a generic error log shown.
- **Poor iterative process:** the lack of specific error feedback forces the user to compare their input with the correct example given to find their mistakes.
- **Separate documentation:** users need to access the online documentation in order to find all valid inputs for a given command, as there is no autocomplete and the help flag does not include every option.
- **Lack of usage artefacts [25]:** the use of these commands does not produce any kind of reference or use history. Users who want to document projects with generated code have to do so themselves, through code comments or external knowledge bases.

These shortcomings are not exclusive to Phoenix and apply to any other Elixir tool that uses Mix Tasks. While that may be acceptable in other contexts, we believe that they are a source of friction in code generation tools, preventing the community from investing in this technology.

1.3 Objectives

The main goal of this project is to elevate Phoenix's code generation features to the same level of usability as the rest of the framework. In order to achieve this, they will be integrated with Livebook to provide a design and documentation platform on which developers can build a collection of Phoenix Mix Tasks into project templates.

There are other tools in the Elixir ecosystem that use Mix Tasks for code generation, bootstrapping, or project configuration, such as the Ash Framework or the Nerves Project. Providing an extensible foundation on which users can build better ways to use them could kick-start the development of this area within the community, giving code generation the status of *first-class citizen* macros and documentation already boast [26]. Therefore, we would like the system to be flexible and extensible enough that parts of it could be reused to improve other software in the ecosystem or even extend its functionality through third-party modules.

Furthermore, we are interested in fostering community interest in this area, so we would like to organise this project following open-source guidelines [27] in order to lower the barrier of entry and promote collaborative development.

1.4 State of the art

Tools for configuring and documenting code generators are not common, even when taking into account other programming languages. However, interactive notebooks and commands for project bootstrapping, which make up the building blocks of this very project, are widespread in the industry. Some notable examples are:

- **Visual Python:** GUI-based Python code generator available through Jupyter Notebooks, one of the tools that inspired Livebook. It offers a series of code components that are visually configured via a series of forms, which are then translated into code blocks in the notebook. There is a wide variety of components, ranging from simple logic to integration of third-party libraries like Matplotlib.

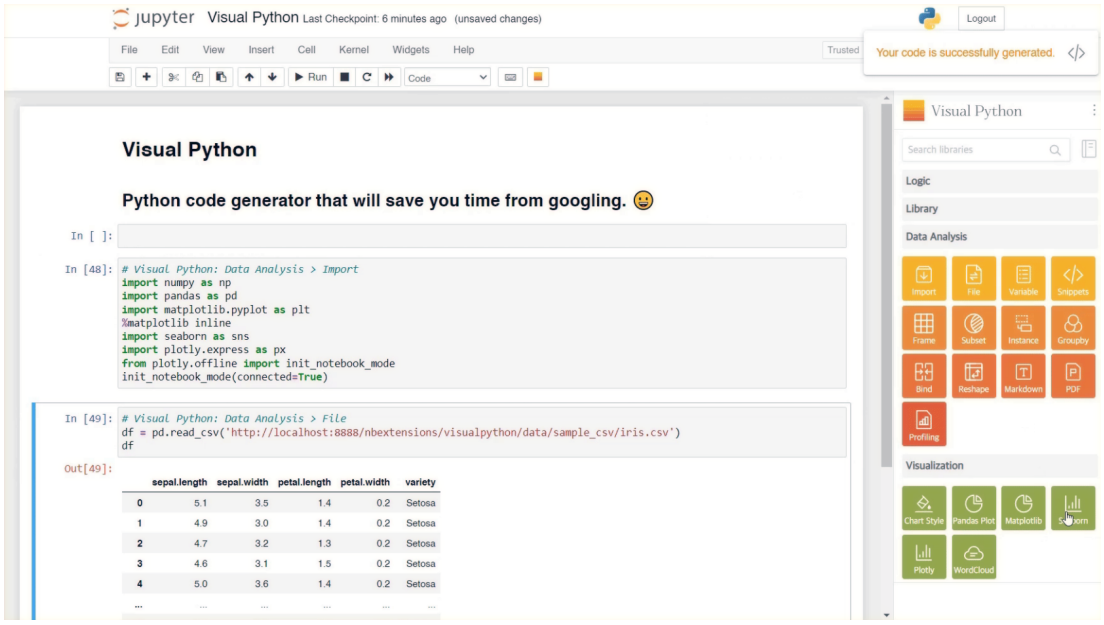

Figure 1.3: Visual Python code generation form

- **phx.new:** simple web application used to generate Phoenix Framework projects. It provides an online form where users can set a series of parameters like the app name or back-end database, and produces the desired application ready for download.

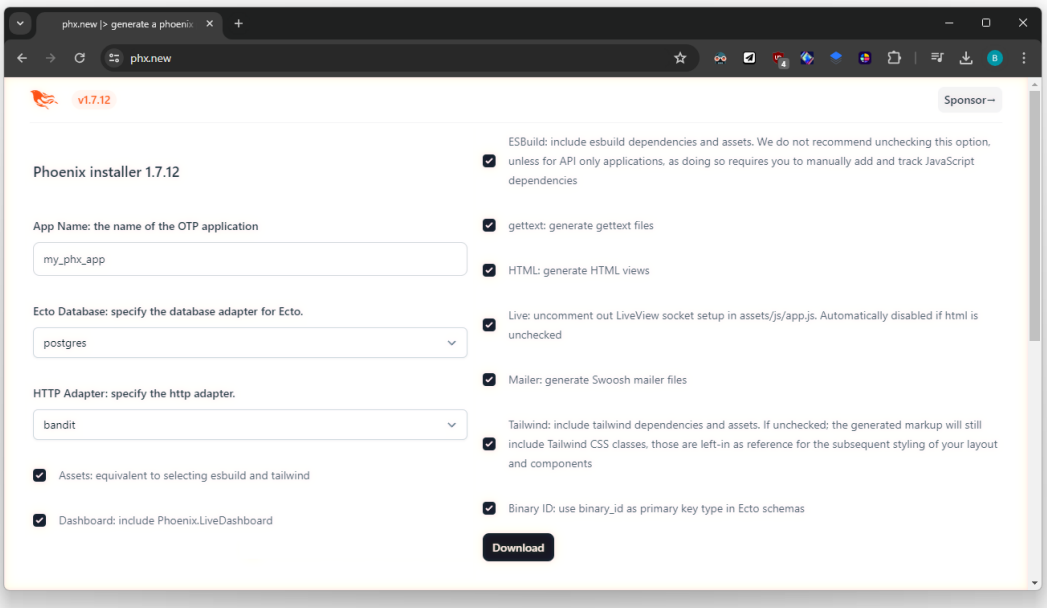

Figure 1.4: Phx.new Phoenix project generator form

- **Eclipse Modeling Tools:** Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model. It allows its users to visually design domain models, graphically analyse existing source code, generate Java code from meta-models, augment existing features, build bespoke environments with ease, or use community-provided packages.

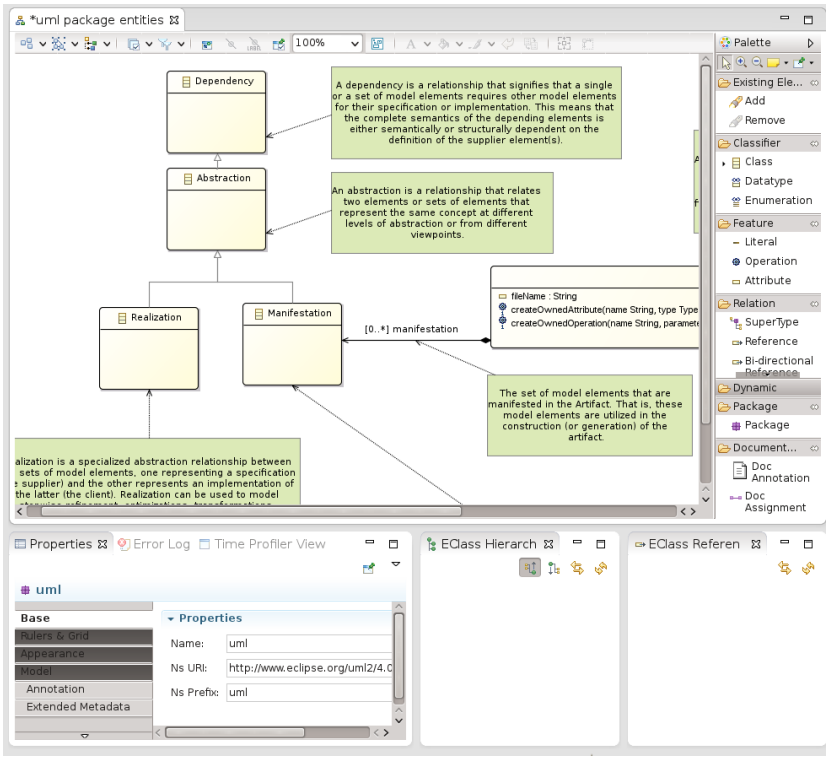

Figure 1.5: Eclipse Modelling Tools

Some key elements of these technologies served as inspiration for our project. Specifically, leveraging existing parameterized commands like phx.new, through a cell-based graphical interface with forms, similar to Visual Python, as well as diagramming and extensibility mechanisms through external packages, akin to Eclipse. The details are described in the following chapters.

Chapter 2
Methodology

In this chapter, we will explain the rules and processes used to develop the project, as well as the guidelines and measures put into place to facilitate the transition into open-source development.

2.1 Work Structure

At the beginning of the project, it proved challenging to define a precise set of tasks due to its uncertain direction and exploratory nature. Additionally, both Phoenix and Livebook are under active development and frequently release breaking changes. We expected the requirements and implementation details to change over time, so we chose an agile approximation. The work was divided into three phases: exploration, prototyping, and stable release, with two-week sprint iterations. At the end of each sprint and phase, we would perform a progress assessment and retrospective in order to correct any emerging issues, react to changing requirements, and set new objectives.

- **Exploration**: the first cycle of the project, dedicated to evaluating the capabilities, limitations, and alternatives to the Livebook and Phoenix Mix commands. During a preliminary meeting, we decided to produce a series of small proof-of-concept programs implementing the basic functional blocks of the system, acting as a viability study.
- **Prototyping**: development of a working prototype without graphical interface. The objectives were to validate the data model, overall system architecture, and use cases. To this end, we introduced the project to the Elixir community through the most popular channels: Elixirforum, Discord and Slack.
- **Initial release**: the longest iteration, aimed at reworking the existing prototype to incorporate user feedback, complete the Livebook integration, produce documentation, and test the system end to end.

2.2 Open-Source workflow

As stated in the project objectives, we wanted to model this piece of work after the typical development processes found in open-source Elixir projects to lower the barrier to entry and encourage contributions. To this end, we chose GitHub to support the entire development process instead of a suite of different products. Both for its status as the de facto platform for open-source communities and for fulfilling the academic requirements of this project, such as progress tracking and visualisation.

We adopted GitFlow [28] as our main workflow for version control. Each feature or bugfix was developed in a separate branch and merged into a general development branch. Eventually, it was merged into the repository's main branch and released with the appropriate tag. However, after the initial release the project will switch to GitHub Flow [29], so contributors will instead fork the original repository, develop the intended feature, and open an external PR. These trigger a set of GitHub Actions to assess passing tests and formatting, as a preventive measure to maintain code quality and prevent regressions.

The system was split into several packages with different repositories. Within each of them, every task was documented with the issue tracker and the corresponding tags in place of a more traditional ticketing system. Any discussion around features or bugs was expected to occur through GitHub comments in the appropriate issue or Pull Request.

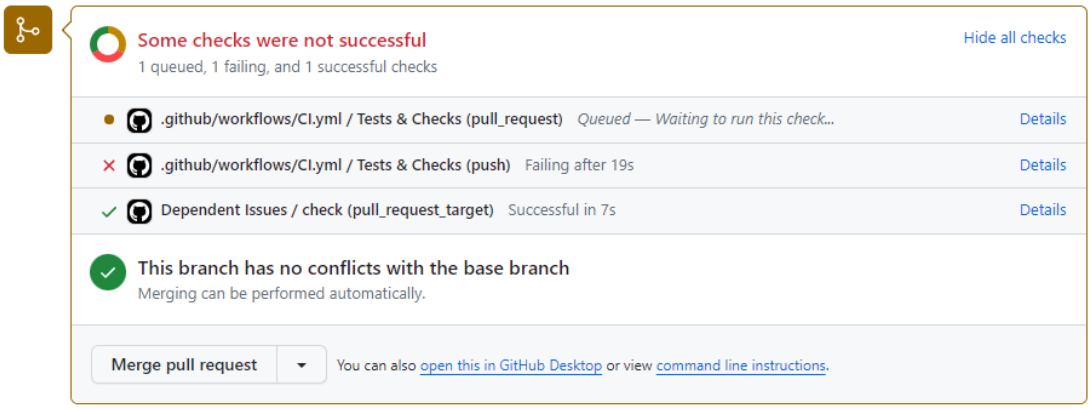

Figure 2.1: Pull request check pipeline

2.2.1 Community Guidelines

As part of the effort to foster contributors, the project makes use of GitHub's Community Standards [30], a set of documents detailing the rules and processes governing contributions and community interactions.

- **README**: main document, displayed on the repository's home page, is used to introduce the project. Typically, it includes a brief description, usage instructions, and links to documentation.
- **Contributing**: describes the process of contributing to the repository, general requirements, and licencing.
- **Code of Conduct**: contains the rules, limitations, and consequences of communication between members of the community.
- **License**: contains the permissions and limitations of the use, modification, and distribution of the repository's contents.
- **Issue and Pull Request templates**: are automatically applied to any issue or PR and delineate the expected style and information to be given.

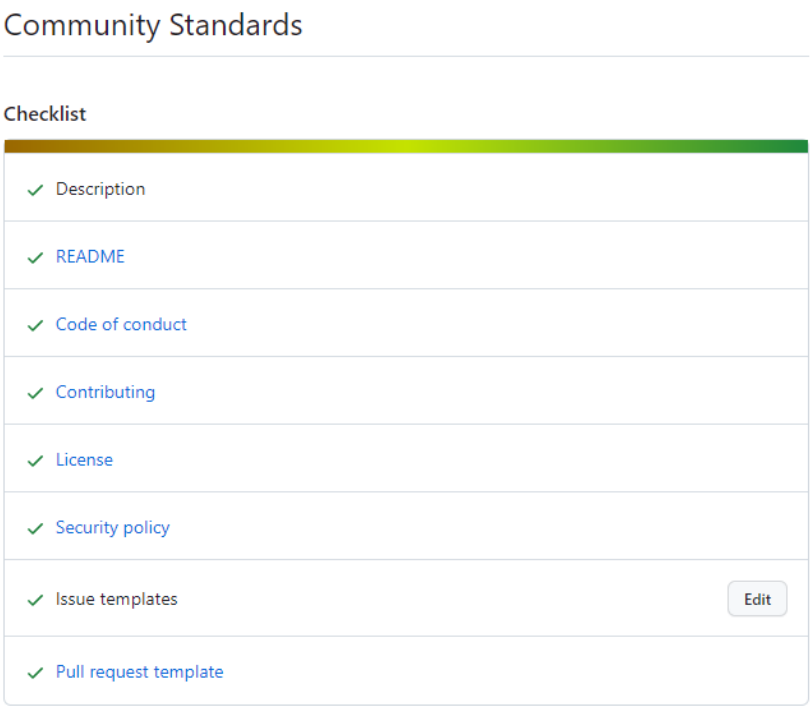

Figure 2.2: GitHub Community Standards checklist

2.2.2 Project Management

Traditional project management is not common in open-source due to the difficulty of coordinating distributed communities of volunteers. However, for the sake of the initial development of this project and progress tracking, we incorporated GitHub Projects as a way to

consolidate the state of the various repositories. This gave us more fine-grained control and information about the repositories' issues and PRs by wrapping them in traditional agile tickets managed through a Kanban-like board. These ticket wrappers allowed us to add context and custom properties to any task, such as task size or priority level, facilitating the use of the built-in visualisation tool.

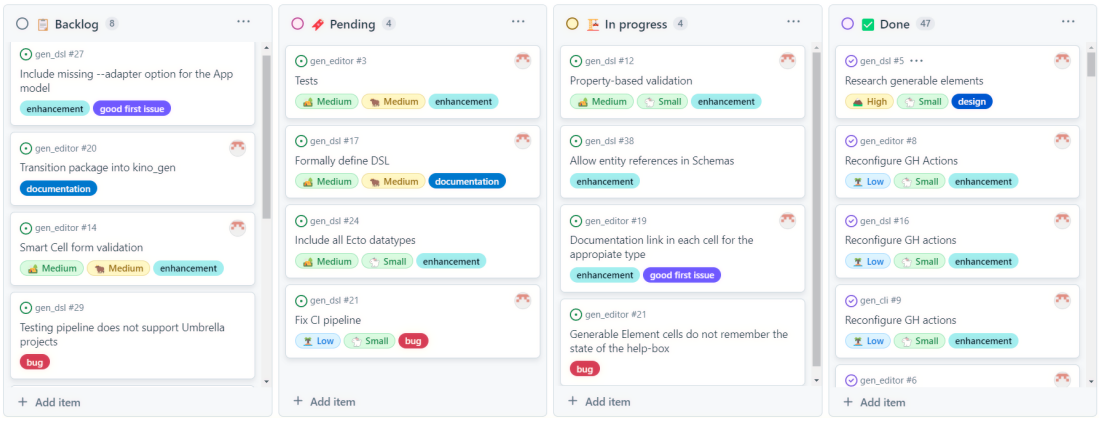

Figure 2.3: GitHub Projects board view during development

One limitation of GitHub Projects is that it does not provide a Gantt chart, but rather a timeline visualisation, as seen in 2.4. However, this project was not intended to be managed like a traditional software development lifecycle and there was no particular focus in task estimation or dependencies, in lieu of a modular approach for issues such that contributors are encouraged to participate without having to coordinate with a development cycle.

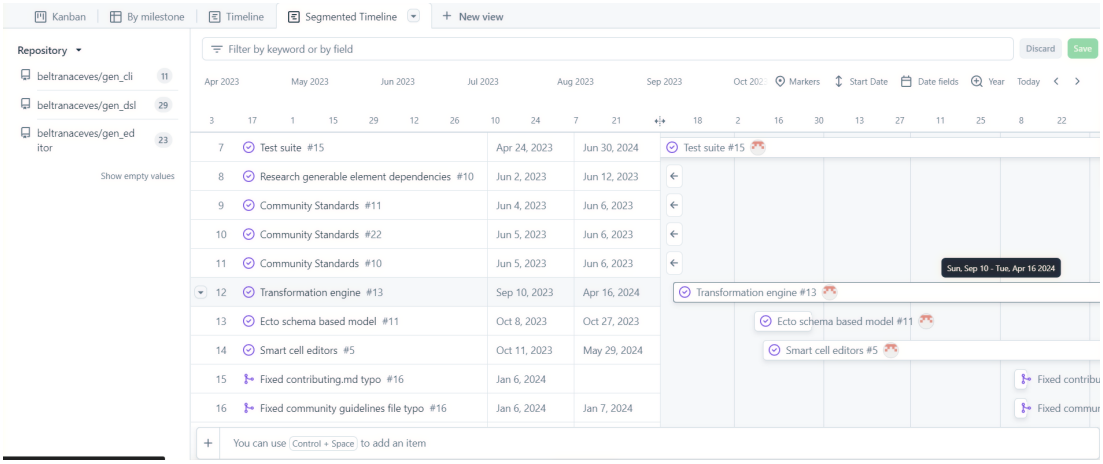

Figure 2.4: GitHub Projects timeline view

Nevertheless, an initial estimation and effort allocation were conducted. The project was projected to require approximately 350 hours, distributed across three phases: exploration (50

hours), prototyping (100 hours), and initial release (150 hours). This estimation was overly optimistic, and the project surpassed the initial deadline of 2023 and continued through 2024. No strict time-tracking was carried, but in number of tickets and code produced, the project took double the initial estimation, around 750 hours.

The project was developed by a team of 2 members: a developer and a project director. The costs associated with human resources have been calculated using the compensation indicated in the collective agreement for engineers in Spain [31]. It assigns a level of compensation to each type of worker depending on their academic achievements and professional function. As such, using the compensation data for 2023, we attribute 28 026,81€ and 21 731,71€ per year, respectively, to a developer and director. For a full-time job of 40 hours per week, 52 weeks per year, the hourly wage comes out to 13,47€ and 10,46€. Given the context of the project being weighted as 12 ECTS credits, we can estimate the director's effort as 8% of the developer's.

Human Resources			
Resource	**Salary**	**Dedication**	**Cost**
Developer	10,46€/h	750h	7 845€
Director	13,47€/h	60h	808,2€
Total			**8 653,2€**

Table 2.1: Human resource cost breakdown

Regarding the material cost of the project, most of the technologies that we used are free and open-source. Services for repository hosting, project management tools, and package distribution were provided by GitHub and Hex. The bulk of the cost then comes from hardware equipment, a 2021 Dell G15 laptop, and an Overleaf standard plan subscription to write this report.

Materials and Services			
Resource	**Unit Cost**	**Units**	**Cost**
Dell G15 (2021)	879,95€	1	879,95€
Overleaf Standard Plan	19€	3	57€
Fly.io Livebook deployment	5€	1	5€
GitHub	0€	1	0€
Hex.pm	0€	4	0€
Hex Docs	0€	4	0€
Erlang OTP	0€	1	0€
Elixir	0€	1	0€
Mix	0€	1	0€
Phoenix	0€	1	0€
Livebook	0€	1	0€
Total			**941.95€**

Table 2.2: Material resource cost breakdown

A timeline of sprints has been reconstructed for the sake of illustrating the project's progress:

- **Sprint 1** (15/11/2022 - 29/11/2022):
  - Implemented a simple project to validate the mechanism for creating Mix Tasks.
  - Experimented with different project structures to find a suitable separation of concerns for future enhancements.
- **Sprint 2** (30/11/2022 - 14/12/2022):
  - Implemented several methods to invoke Mix Tasks programmatically.
  - Began researching the pros and cons of different invocation methods.
- **Sprint 3** (06/02/2023 - 22/02/2023):

  - Tried to implement a proof of concept Mix Task chain by creating a Phoenix project and requesting generation of an Html resource, unsuccessfully.
  - Researched how dependencies are loaded into the Elixir runtime.

- **Sprint 4** (23/02/2023 - 07/03/2023):
  - Finished research Mix Task invocation methods and started preparing a data model for generable elements.

- **Sprint 5** (08/03/2023 - 22/03/2023):
  - Completed the data model using structs and a system to read them from a JSON file.

- **Sprint 6** (24/03/2023 - 07/04/2023):
  - Implemented a working procedure for generating Phoenix projects with arbitrary generable elements using a mix of method for the current process and new ones.

- **Sprint 7** (10/04/2023 - 26/04/2023):
  - Redefined the data model using Exconstructor to load elements into memory after rejecting the idea of implementing a custom DSL with NimbleParsec.

- **Sprint 8** (27/04/2023 - 12/05/2023):
  - Developed a simple UI cell to configure the App element using Kino to validate the use of Livebook as a base for the system.

- **Sprint 9** (15/05/2023 - 30/05/2023):
  - Finished the exploration phase and started developing a prototype.
  - Split the project into three modules: Gen DSL, Gen CLI, Gen Editor.

- **Sprint 10** (31/05/2023 - 14/06/2023):
  - Started to migrate the data model to Ecto schemas to enjoy the benefits of a more complete abstraction.
  - Identified the problem around functional dependencies between elements.

- **Sprint 11** (15/09/2023 - 29/09/2023):
  - Developed a CLI wrapper for Gen DSL in Gen CLI using OptionParser and Mix Task invocations.

  - Packaged said wrapper as a Mix archive and published to Hex.pm.

- **Sprint 12** (02/10/2023 - 17/10/2023):
  - Developed a simple Smart Cell to validate the return on investment for the effort required.
  - Finished migrating the data model to Ecto schemas.
- **Sprint 13** (18/10/2023 - 02/11/2023):
  - Changed the element generation method from pattern matching to global selection of functions with atom over Elixir's global module namespace in Gen DSL.
  - Started defining the changes needed to support a plugin system.
- **Sprint 14** (15/01/2024 - 30/01/2024):
  - Implemented a runtime dependency injection system for Gen DSL to support the plugin system.
- **Sprint 15** (31/01/2024 - 13/02/2024):
  - Ported Gen CLI distribution bundle to mix escript to reduce complexity when handling multiple levels of nested VM calls and their dependencies.
- **Sprint 16** (13/02/2024 - 28/02/2024):
  - Finished designing the first iteration of the plugin system.
  - Started implementing Smart Cell-based configurator for every generable element.
- **Sprint 17** (14/03/2024 - 01/04/2024):
  - Finished implementing the first iteration of Smart Cells.
  - Started implementing nested Erlang VMs to handle plugin dependencies.
- **Sprint 18** (02/04/2024 - 16/04/2024):
  - Reworked the graphical configurators to solve the functional dependency resolution problem.
  - Finished the VM management system to complete the plugin system.
- **Sprint 19** (17/04/2024 - 01/05/2024):
  - Started writing the report for the project.
  - Researched novel methods for testing generated projects.

- **Sprint 20** (02/05/2024 - 17/05/2024):
  - Implemented an involved system using property-based testing to validate Gen DSL project generation.
- **Sprint 21** (20/05/2024 - 04/06/2024)
  - Continued writing the project report.
  - Reworked Gen DSL's generator functions to be POSIX compatible after a cloud deployment failed catastrophically.
- **Sprint 22** (05/06/2024 - 20/06/2024):
  - Completed all specified project requirements.
  - Finished the project report.

Chapter 3

Requirements Analysis

In this chapter we will describe the requirements the system aims to fulfil, the actors involved, and the use cases.

In order to accomplish the aforementioned objectives, the project must strive to fulfil a series of requirements. They will lay the foundation for further design and implementation decisions and establish the acceptance criteria when evaluating the final system.

3.1 Actors

We identified two actors through our system requirements analysis, the authenticated user who uses the system to document or configure code generation, and a developer who extends the functionality of the system. Livebook notebooks have complete access to the underlying system through its file system and arbitrary Elixir code execution; for this reason, the system requires users to authenticate themselves and does not offer functionality to anonymous users.

3.2 Functional Requirements

These are the behaviours and functionality which form the core capabilities that the system must support to achieve the outlined objectives and meet the needs of its users.

- **FR-01 User**: Log in as a user on Livebook.
- **FR-02 User**: Manipulate Livebook notebooks. Includes creation, deletion, edition, and change of directory.
- **FR-03 User**: Manipulate generable element graphical configurators in a Livebook notebook.

- **FR-04 User**: Change the type of generable element and retain user input.
- **FR-05 User**: Validate the input of the graphical configurator.
- **FR-06 User**: Document generable elements through text, image, or diagram cells in a Livebook notebook.
- **FR-07 User**: Access the documentation of generable elements.
- **FR-08 User**: Build generable element specification.
- **FR-09 User**: Download generable elements' specification.
- **FR-10 User**: Execute code generation with specification in the Livebook host system.
- **FR-11 User**: Download generated code as compressed folder.
- **FR-12 User**: Execute code generation through arbitrary terminal.
- **FR-13 User**: Persist generable element configurator values in notebook file.
- **FR-14 User**: Persist generable element configurator outputs in notebook file.
- **FR-15 User**: Validate generable element configurator parameter syntax.
- **FR-16 User**: Access in-editor documentation for each generable element.
- **FR-15 Developer**: Allow dependency injection into the code generation system.
- **FR-16 Developer**: Allow behaviour injection into the code generation system.
- **FR-17 Developer**: Allow third-party systems to interact with generable element specification within notebooks.

3.3 Non-Functional Requirements

In contrast to functional requirements, these specify qualitative aspects and criteria that are used to assess the operation of the system.

- **NR-01 Security**: require user authentication to access any part of the system. The system can include internal design documents and project generators that might be required to be private, and third-party extensions can access the host system, so execution should be only possible for those with permissions.
- **NR-02 Integration**: software must integrate every generable element from the Phoenix Framework.

- **NR-03 Suitable for version control**: both the graphical editor and generable element specification must be easy to check into version control and allow for semantic search between versions in order to facilitate iterative processes.
- **NR-04 Live collaboration**: the system must allow several users to collaborate and edit simultaneously any generable element editor.
- **NR-05 Extensibility**: developers must be able to easily add graphical configurators and types of generable elements through external packages.

As we will explain in the following chapter, some of these requirements were fulfilled by our technology choices instead of having to implement it ourselves. In particular, Livebook provides compliance for FR-01, FR-02, FR-06, FR-13, FR-14, NR-01, NR-03, NR-04, and Phoenix provides the foundations for NR-02. The complete breakdown is shown in 3.1

Requirement	Livebook	Phoenix	Project
FR-01	✓		
FR-02	✓		
FR-03			✓
FR-04			✓
FR-05			✓
FR-06	✓		
FR-07			✓
FR-08			✓
FR-09			✓
FR-10			✓
FR-11			✓
FR-12			✓
FR-13	✓		
FR-14	✓		
FR-15			✓
FR-16			✓
FR-17			✓
NR-01	✓		
NR-02		✓	
NR-03			✓
NR-04	✓		
NR-05			✓

Table 3.1: Requirement compliance summary

Chapter 4

Architecture and design

In this chapter, we will propose a system and describe the overall architecture and details of its design.

4.1 Motivation

To address the aforementioned shortcomings of Phoenix's `mix phx.gen` Tasks, we propose a system that reconsiders how code generation is provided to developers. The current process involves sequences of commands similar to the following:

1. `mix phx.new AppName`, to generate an empty Phoenix project with default scaffolding.
2. `mix phx.gen.auth Accounts User users`, to add authentication and protected routes.
3. `mix phx.gen.live Content Messages messages author_id:references:user recipient_id:references:user context:string`, to add soft real-time routes for CRUD operations over a message resource.
4. `mix phx.gen.cert`, to create and sign a certificate for local development.
5. `mix phx.gen.release --docker`, to build a release for the project and an accompanying docker file for deployment.

Although this method is executed in several steps, it couples design and implementation into a single phase, making it difficult to make a distinction between the two in order to provide an adequate environment and tooling for each. As illustrated in 4.1, our proposal is to split the process into two workflows:

- **Graphical Configuration Workflow**: Using Livebook's graphical interface, this workflow would allow users to configure project templates interactively that represent se-

quences of Mix Tasks commands, and document them through diagrams, design decisions and motivations, code snippets with syntax highlighting for public APIs, and more. Through Smart Cells, users can define various generable elements, laying a foundation on which we can add in-editor documentation, auto-complete, collaborative editing, validation for parameters and element dependencies, as well as producing design artifacts suitable for version control and distribution.

- **Command-Line Interface (CLI) Workflow**: enables the execution of code generation mix tasks from a template from a single command, reducing repetition and minimising user error. It also allows third-party development of custom template creation systems.

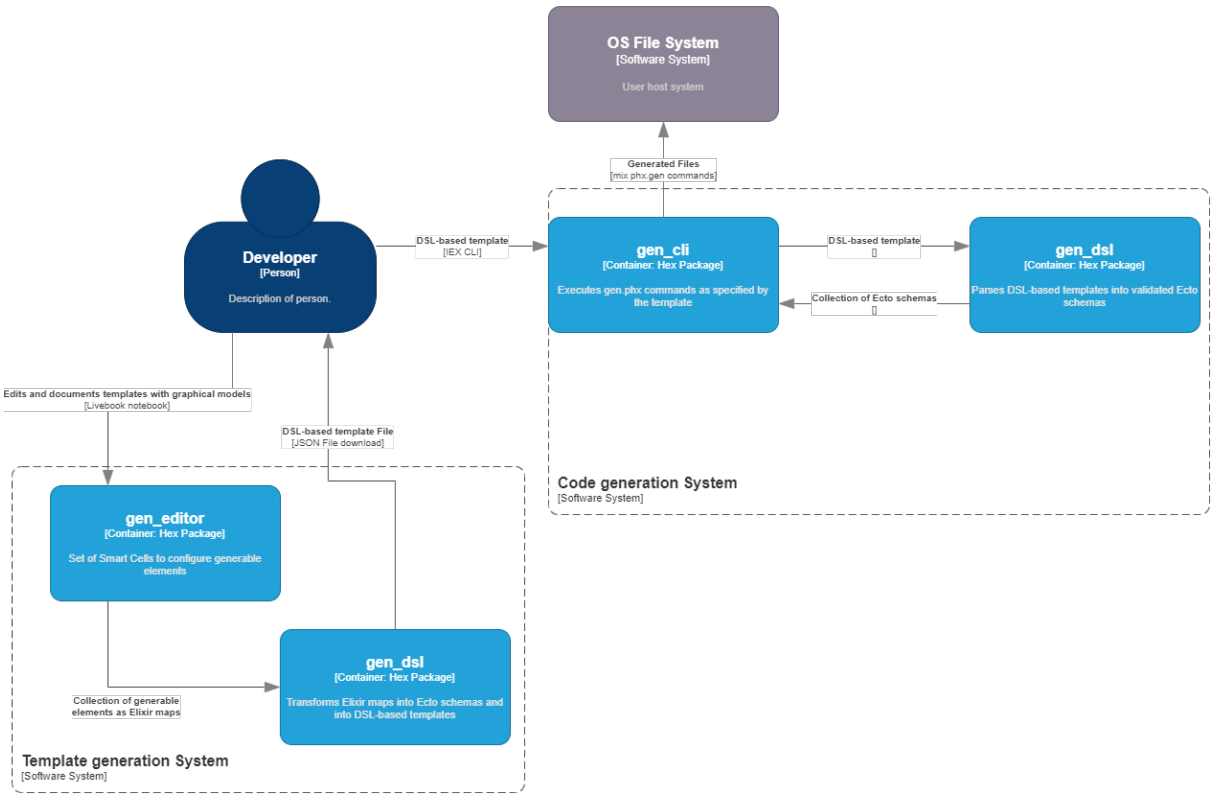

Figure 4.1: Container view of a C4 diagram of the system and its workflows

To mitigate the inconvenience of context switching in iterative design and to accommodate developers who prefer a unified process, the CLI workflow would be integrated into Livebook. This integration would offer a cohesive environment and eliminate the need for local system installations, in favour of a shared instance to encourage collaborative work.

The system has been conceived as a collection of packages segregated by responsibilities, forming a toolchain that integrates with Phoenix Framework and Livebook to achieve

its objectives. We illustrate it in 4.2 using the C4 model, an abstraction-first approach to diagramming software architecture that provides different levels of detail depending on the level of simplification: context, container, component and code.

These packages represent modules in charge of interacting with different external environments, while keeping a uniform interface to minimise coupling for users that wish to use only a portion of our system. Specifically, there is a package responsible for graphically configuring project templates that interacts with Livebook, another package parses such templates and interacts with Phoenix's code generation functions, and a third package provides a terminal interface that interacts with the user and the underlying systems. Their relationship can be seen in 4.2.

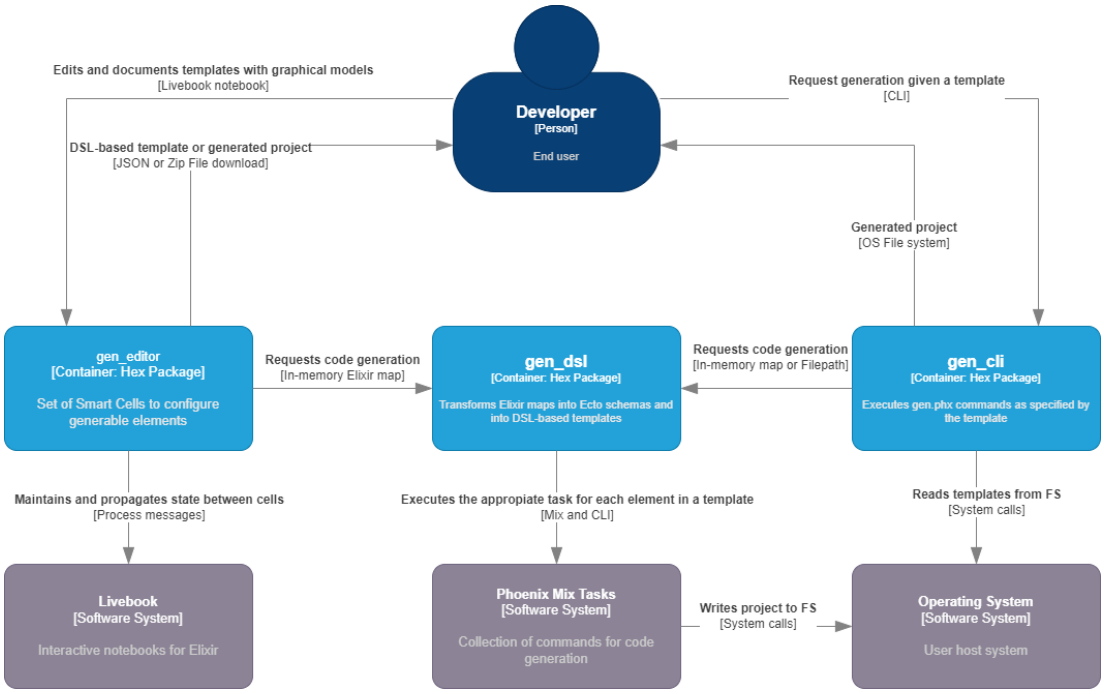

Figure 4.2: C4 representation of the toolchain interfaces

When choosing an architecture and designing the core elements of the system, the integration requirements had a high impact. In particular, Livebook forced a distributed client/server architecture for each of the graphical configurators since they reside each in their own process and distribute data not present in the server. The rest of the packages have been implemented with simpler client/server architectures, with the possibility of transitioning to a leader/worker scheme if code generation performance becomes an issue.

4.2 Gen DSL

The package in charge of defining the generable element data model, parsing and serialising project templates, as well as interacting with Phoenix Mix Tasks. It contains modules for each one of the 15 types of generable element currently provided by Phoenix:

Table 4.1: List of Phoenix generable elements

Element	Description
App	New Phoenix started project
Schema	Ecto-based database schema and migrations
Auth	Authenticacion system based on an Ecto resource
Cert	Locally signed certificate for local development
Channel	PubSub channel for soft real-time communication
Embedded	Ecto embedded schema for out-of-database data manipulation
Html	Controller and view templates with CRUD operations for an Ecto schema
Json	Controller endpoint to serve JSON through a REST API
Live	Controller and view templates with CRUD operations for an Ecto schema using LiveView for real-time updates
Notifier	Email notifier with topic-based messages
Presence	Real-time user route presence channel
Release	Release files and dockerfile
Secret	N-length random secret
Socket	Route-based lower level websocket

Each of these elements has additional mandatory and optional parameters to specify the generated contents, as well as dependencies between them. For example, the Html element generator also generates the schema and migrations for the resource it exposes, so instead of using the schema command and then the Html command, only the latter is needed. This creates the need to analyse dependencies between generable elements such that they do not

step over each other, resulting in a failed execution. This responsibility does not lie in this package, and it expects correctly formed project templates.

Additionally, it is in charge of loading external dependencies specified in the templates in the form of plugins, allowing external developers to extend the available generable elements or otherwise project manipulation functions.

4.3 Gen CLI

The package in charge of providing template code generation from the terminal. Its main goal is to be easily installed with Elixir-provided tools and to allow repeatable project generation from files without redeclaring its specification. It contains a command parser module that invokes Gen DSL and exposes a mix task to the end user. In contrast to the rest, this package has been bundled with an Elixir runtime and packaged as an escript, as show in 4.3, to allow local installation on end-user equipment.

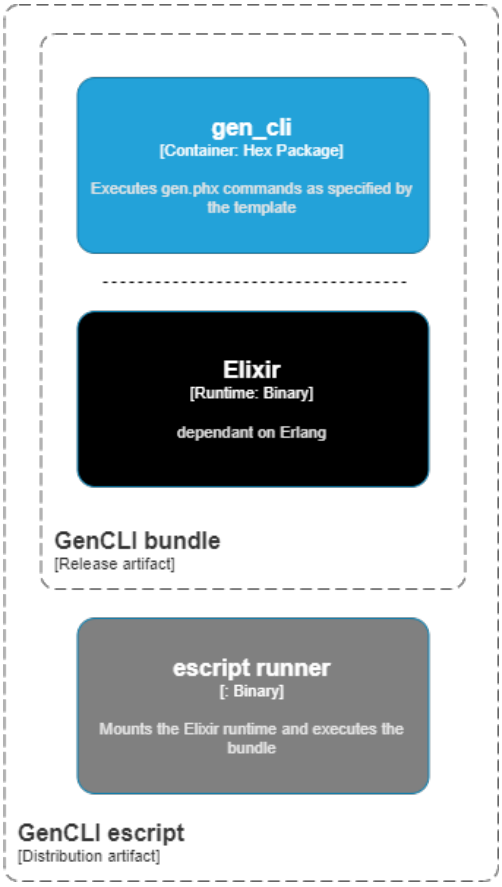

Figure 4.3: Distribution artifact structure for Gen CLI

4.4 Gen Editor

The package in charge of extending the behaviour of Livebook to provide graphical configurators of generable elements. It uses Livebook's Smart Cell abstraction to interact with the user and notebook, which forces the package to follow its interface requirements. It is considered a distributed client/server architecture since every client (Smart Cells) runs in a different thread, as well as the main Livebook cell. Each cell can also act as a server when broadcasting data changes in its state.

4.5 File format

The code generation templates are JSON files with a specific structure. They are parsed into Elixir maps using Poison and then cast into Ecto Schemas, allowing us to take advantage of Ecto's changesets for data validation and error messages for user feedback. While the system allows for templates with non-standard elements, it enforces the existence and structure of the following keys:

- *Dependencies*: optional packages to be installed via Mix.install/1 to import third-party functionality.
- *Pre-tasks*: preliminary functions to perform before code generation.
- *Generable elements*: list and specification of Mix Tasks to generate the project.
- *Post-tasks*: clean up functions to perform after code generation.

4.6 Extensibility

Third-party developers can extend the system's functionality in two ways. They can develop custom Smart Cells for Livebook to modify the template state, which is kept public for this very purpose. Alternatively, they can implement a Gen DSL plugin and publish it using any method that Mix.install/1 can process. Once added to a template's dependencies, Gen DSL will automatically load the plugin, enabling the generation of new elements. These mechanisms can be seen in 4.4 and 4.5.

To facilitate and encourage the development of Gen DSL plugins, a Mix Task has been made available to generate an empty plugin project that fulfils the minimum requirements of the interface.

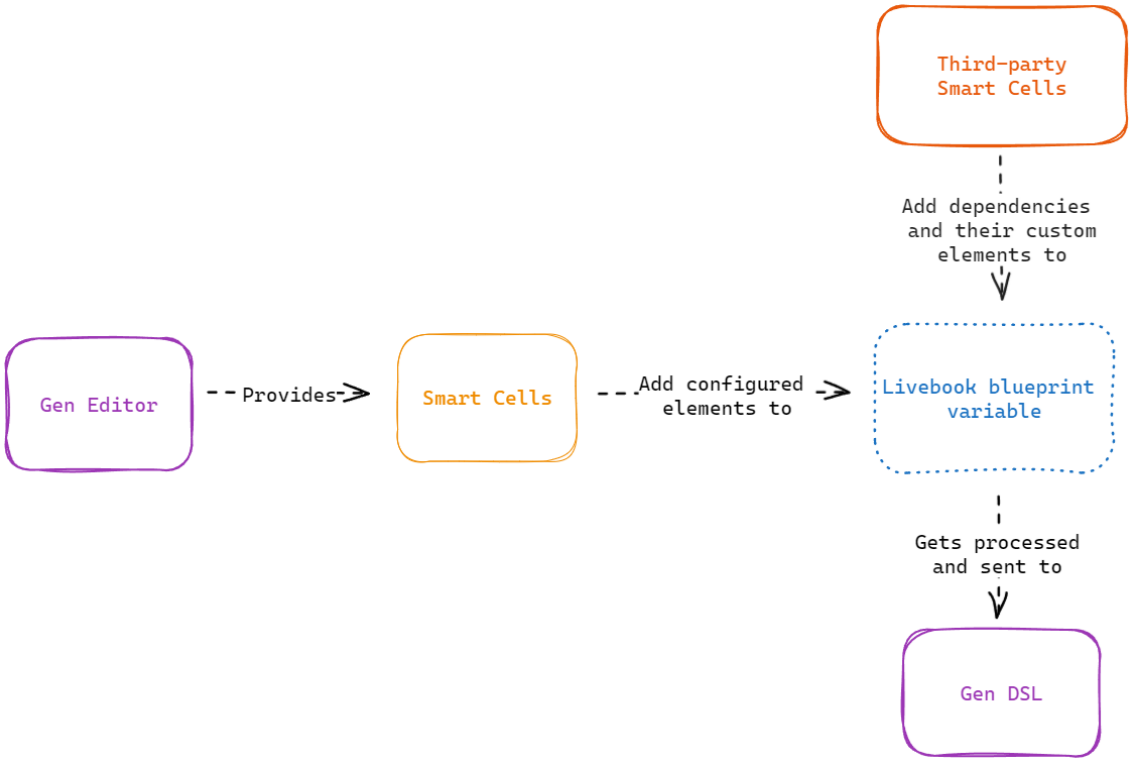

Figure 4.4: Data flow in Livebook for Gen Editor

Although the system could have been implemented as a monolithic application with a layered architecture, we decided early on to split each layer into a separate package. By separating the application into a series of microservices that act as reusable code components, we facilitate the progressive adoption of the toolchain and support use cases outside the scope of our project. Hiding implementation details with interfaces between components helps us reduce coupling and allows users to extend the functionality of the system by developing modules that fulfil such interfaces.

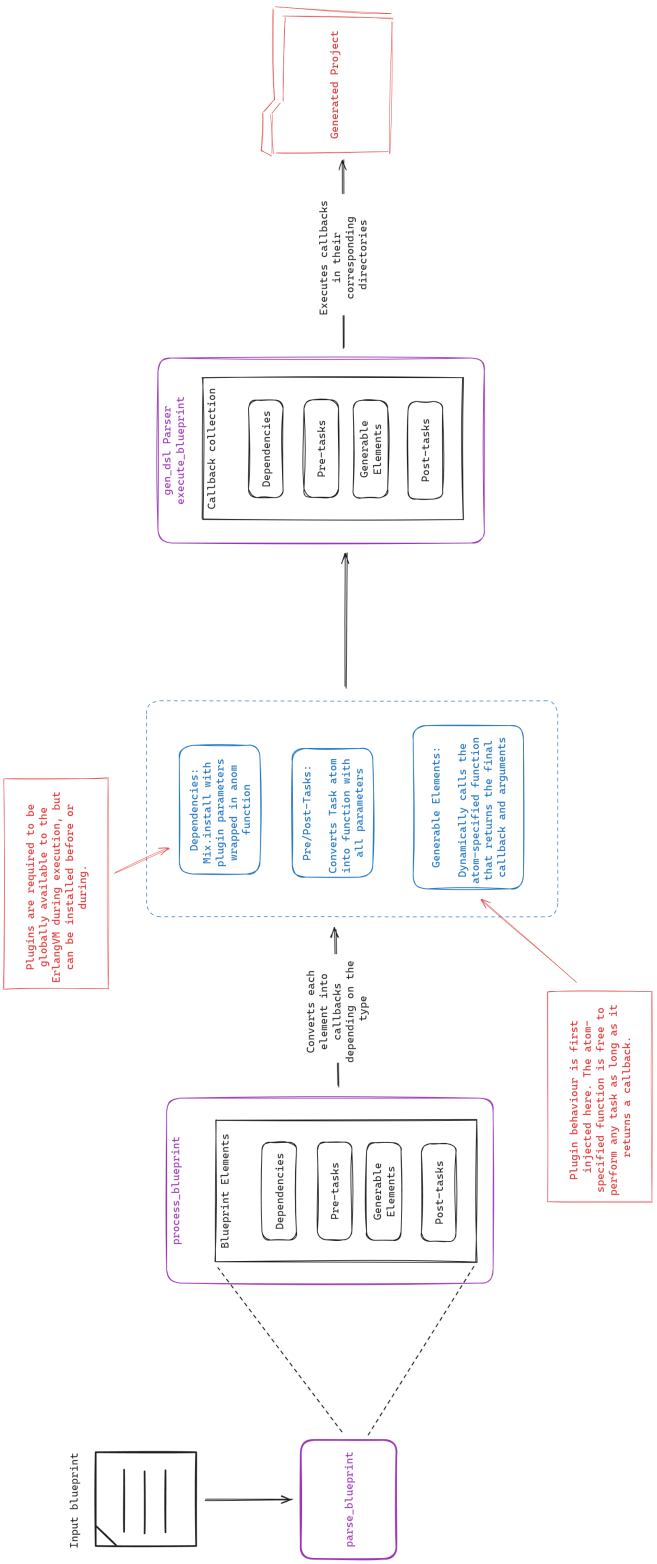

Figure 4.5: Data flow and execution step for Gen DSL

Chapter 5
Implementation

In this chapter we will explore the different hurdles encountered during the implementation of the system and the design decisions we made along the way.

As we mentioned previously, given the exploratory nature of the project, it was divided into three distinct phases: exploration, prototype, and initial release. Such separation allowed us to continuously adjust to the rapidly changing Elixir landscape, since the tools we wanted to interact with were in active development. It also provided us with the opportunity to validate ideas, collect feedback, redefine our objectives, and respond to changing requirements. This structure proved remarkably useful, as throughout the development process we encountered several unviable avenues that forced us to pivot, due to both technical limitations and the appearance of better alternatives.

5.1 Exploration

The objective of the first phase was to validate the mechanisms needed to establish the foundation on which to provide a better developer experience for those using code generation.

The initial hurdle was figuring out how to programmatically invoke Mix Tasks. Mix is a build tool that ships with Elixir that provides tasks for creating, compiling, testing your application, and managing its dependencies. It is intended to be invoked through the terminal during development as it is not bundled with compiled applications and thus not available in production. Most code generation capabilities are distributed through Mix Tasks, so being able to execute them from our program was critical to the project. There are multiple ways to do so, but most of them were not suitable for our purposes:

Current process	New process
Mix.Task.rerun/2	*Port.open/2*
Mix.Tasks.Phx.Gen.New/2	*Kernel.send/2*
File.cd/2	*Mix.shell().cmd/1*
System.cmd/1	
System.shell/1	
:os.cmd/1	

Table 5.1: Mix Task programmatic invocation methods

These methods can be split into two groups, those that call Mix from the current process and those that spawn a new one. That distinction is very significant due to the way dependencies are loaded. Elixir runs on the Erlang Virtual Machine, which means that whenever a Mix command is invoked, it sets up a new VM, loads any dependencies present in the local archive, any Elixir project present in the current directory, and sets the context application. Once this process has taken place, no other dependencies can be loaded without tearing the VM down, and the context application cannot be changed. In the context of using Phoenix Mix Tasks, those methods that use the current processes limit the use of code generation, and only the new project generator works, since the dependencies to execute the rest of the available commands cannot be loaded into the current VM. Even if that was not the case, the context app still cannot be changed, and those tasks would fail during execution. On the other hand, those methods which do spawn another process setup a VM from scratch each time, allowing us to reload dependencies and chain together code generation tasks at the expense of very short-lived sessions. From the remaining options, we decided to use `Mix.shell().cmd/1` since it provides a simple high-level interface.

The next step was to define a way to load configuration files into memory. Neither Elixir nor Erlang provide parsers for common file formats such as JSON, TOML, or YAML, so we brought in an external dependency, Poison Poison, to avoid rolling out our own implementation. We had the option of implementing our own syntax with NimbleParsec, but the benefits of a more terse DSL did not justify the effort required. Also, as of Erlang 27 [32], native JSON support has been included. Finally, we needed a mechanism to add a terminal command to execute our system. We chose to build a Mix Task, which allowed us to implement it in Elixir and not rely on another runtime, and packaged it as a Mix archive so that users could install it globally on their systems.

At this stage, we also took some time to explore the possible the visual elements we could provide as a way to configure code generation tasks. Domain-specific languages, interactive diagrams, as well as drag-and-drop interfaces were considered, such as DBML or DBDiagram.io. However, they were ultimately abandoned in favour of simple forms and marked as outside the scope of this project, due to the considerable effort to implement them and the lack of existing libraries in the Elixir ecosystem at the time. Even with the reduced scope, we did not have a clear path to develop these graphical configurators. Kino is a library developed by the Livebook team to easily add and extend UI elements. It provides `Kino.Controls`, a series of Elixir primitives for the most common web elements, such as grids, buttons, images, text inputs, etc, as well as a more involved interface to build our own using Html, CSS, and JavaScript. For the time being, we decided to use the simpler primitives to implement the App generable element, which bootstraps new Phoenix projects, as seen in 5.1.

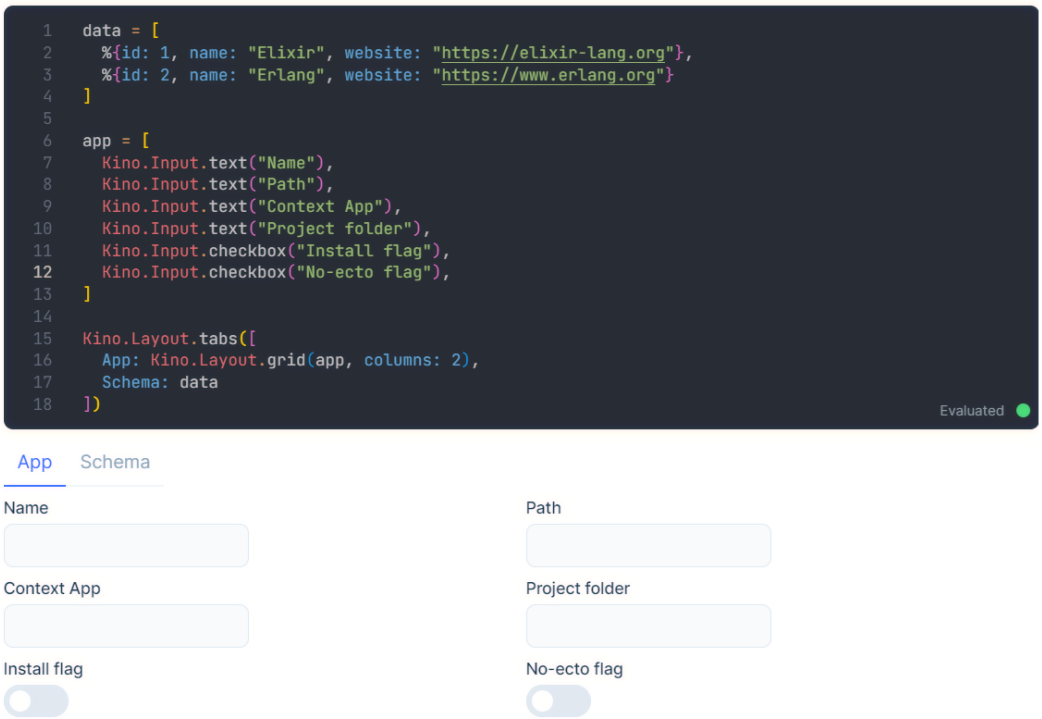

Figure 5.1: Example of UI composed of Kino.Controls and Inputs

5.2 Prototype

Once we had validated these building blocks, we entered the second phase and focused our efforts on implementing a prototype. At that point, we decided to modularize the project into

three separate packages with different responsibilities. As stated in the architecture chapter 4.1, we split the project into `Gen DSL`, `Gen CLI`, and `Gen Editor`.

5.2.1 Gen DSL

There were several options when deciding how to implement an initial data model. Since Elixir is a functional programming language and organises its code in modules and functions, we could not model our data as a typical object like we would in object-oriented languages. Our options were simple dynamic maps, structs, and Ecto schemas. When parsing template configuration files into memory, we wanted to use a data structure that afforded us flexibility since the data model had not been established yet, as well as autocomplete features for ease of development. We settled on using structs, since they provide us autocomplete as opposed to maps and a low barrier of entry compared to Ecto schemas, while keeping changes to the model manageable.

When parsing template files and casting them into elements of the data model, we experimented with adding the Exconstructor macro to the struct definitions. It aided with validation, but we ended up opting for the basic method provided by Poison, `decode!(as: %Elements)/1`, for the sake of simplicity and reducing dependencies.

During this process of exploring the depth of model validation, we discovered a bigger problem with using multiple Phoenix tasks. The elements these tasks can generate are not mutually exclusive and an element can be composed of several others. For example, the command for the `Html` element, `phx.gen.html` takes as parameters a `Context` and a `Schema`, which in turn have their own `Context` and `Module`. Likewise, generating elements that overlap with existing ones fails if not separated into a different namespace. Meaning, several elements can and should be generated from a single command if possible, to avoid unnecessary namespacing.

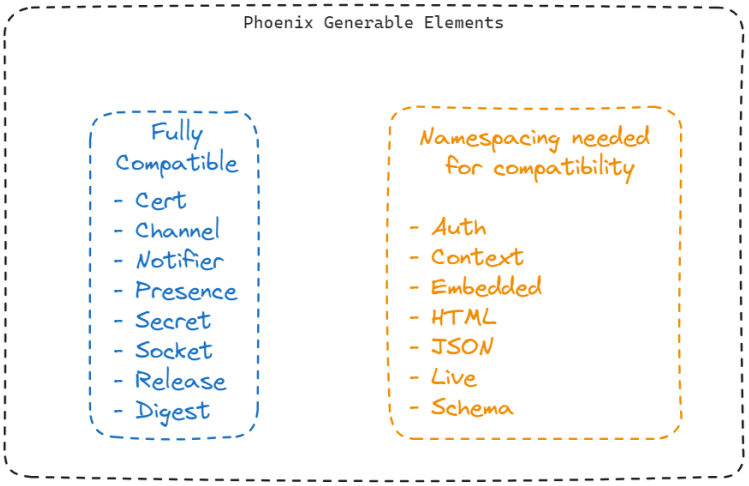

Figure 5.2: Classification of generable elements

As a result, when generating a template, the functional dependencies between elements must be taken into account to prevent resources from overlapping, be it by introducing artificial namespaces or combining multiple elements into a single command. During this phase of development it was deemed out of scope and pushed further down the line, anticipating significant effort to solve the dependencies or find an alternative execution order.

5.2.2 Gen CLI

At this point, we had a working Mix Tasks to invoke the rest of the system, available only on the project directory. However, we wanted to replicate the seamless installation process other libraries provide, in which a user can access installed Mix Tasks throughout the entire system. For example, a developer starting with the Phoenix Framework would run to download, compile, and install the *phx_new* package in their Elixir environment, allowing them to run `mix phx.new` in any directory to bootstrap an empty Phoenix project with all the necessary dependencies.

In order to achieve that, we took advantage of `mix archive`, a local repository of Elixir packages that are loaded from memory into the VM when mix is invoked. Gen CLI needed to be packaged and published in Hex, the Erlang ecosystem package manager, as a prerequisite to enable the command `mix archive.install hex gen_cli` as the primary method for installing Gen CLI. We only needed to create a Hex account and add metadata to our `mix.exs` file, similar to `package.json` or `pom.xml` in the JavaScript and Java ecosystems, respectively. A minimal example would be:

```
1 defmodule Postgrex.MixProject do
2   use Mix.Project
3
```

```
  def project() do
    [
      app: :postgrex,
      version: "0.1.0",
      elixir: "~> 1.0",
      build_embedded: Mix.env == :prod,
      start_permanent: Mix.env == :prod,
      description: description(),
      package: package(),
      deps: deps(),
      name: "Postgrex",
      source_url: "https://github.com/elixir-ecto/postgrex"
    ]
  end

  defp description() do
    "A few sentences (a paragraph) describing the project."
  end

  defp package() do
    [
      # This option is only needed when you don't want to use the
    OTP application name
      name: "postgrex",
      # These are the default files included in the package
      files: ~w(lib priv .formatter.exs mix.exs README* readme*
    LICENSE*
                  license* CHANGELOG* changelog* src),
      licenses: ["Apache-2.0"],
      links: %{"GitHub" =>
    "https://github.com/elixir-ecto/postgrex"}
    ]
  end
end
```

5.2.3 Gen Editor

Previously, we had used `Kino.Controls` to implement a basic configurator, but soon realised that this method was not suitable for our needs. We had no way of selecting a configurator based on the element type, and it required us to distribute interactive notebooks with pre-defined code cells instead of an Elixir package to extend Livebook. Instead, we decided to develop our own Smart Cells, which can be easily installed in any notebook and are accessed through a simple dropdown. They require significantly more effort and run each in their own process, but are dramatically more customisable thanks to their Html, CSS, and JavaScript

implementation. Kino exposes the `Kino.SmartCell` interface to add custom cells, as seen in 5.3, and required us to implement a `to_attrs` method used to persist its content in the notebook file, and `to_source`, used to translate a cell into Elixir code for execution. It also provides an interface to handle message passing between the JavaScript context of each cell and its Elixir counterpart, allowing us to propagate the state between users modifying the same notebook.

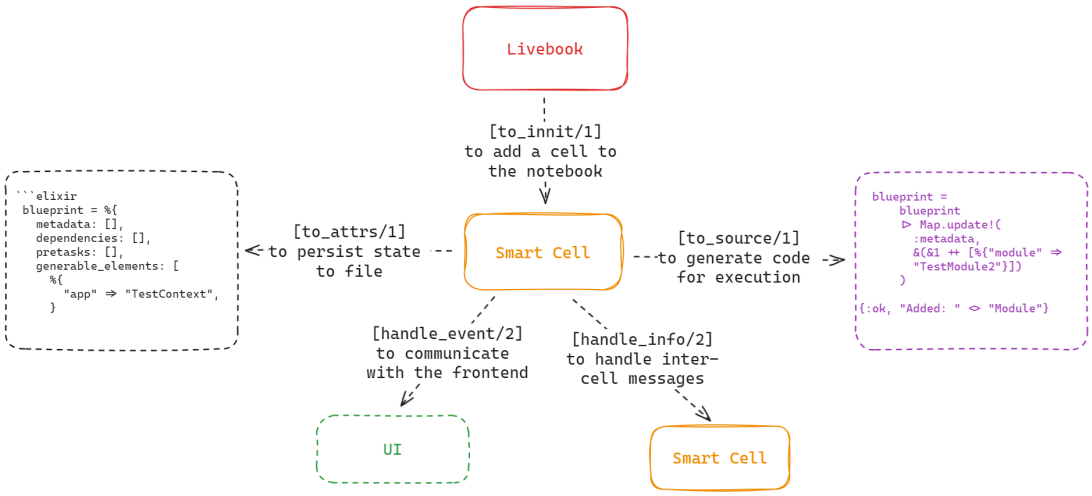

Figure 5.3: Callbacks provided by Kino to manage Smart Cells

5.3 Release

By this point, we had a working version of the CLI workflow 4.1. The Configuration 4.1 workflow had been delayed by changes introduced in Livebook and Phoenix during the previous phase. To prevent that from happening again, we decided to freeze the versions of both dependencies our system would support, v0.11 and 1.7.10, respectively, and continued development:

5.3.1 Gen DSL

Thanks to a more stable data model and a better understanding of its uses, we could finally transition to a more robust data layer at the expense of flexibility. We rewrote every generable element model as an Ecto Schema, achieving much better validation, casting error feedback, and nested element embedding. A complete data layer implementation showcase is available at A.

In an effort to make our code more reusable, we reworked how templates are processed, as seen in 5.4. Instead of pattern matching a `generate_element/2` function given a generable

element type, we switched to applying a function over the element as data. The advantage of this is that, thanks to Elixir's global module namespace that allows any function to access any other module without imports, we are free to structure our model however we want without leaking the abstraction into the template format. With that, template elements simply include an atom that provides the function with which they should be processed. We realised that this meant that if we could load external modules at runtime, other users could develop their own functions to process and generate elements. This proved a fantastic opportunity to fulfil the extensibility requirement of Gen DSL, so we formalised an interface for plugins and added a method for loading external dependencies into the global module namespace, at the cost of significant complexity and effort. More details will be given in chapter 7.

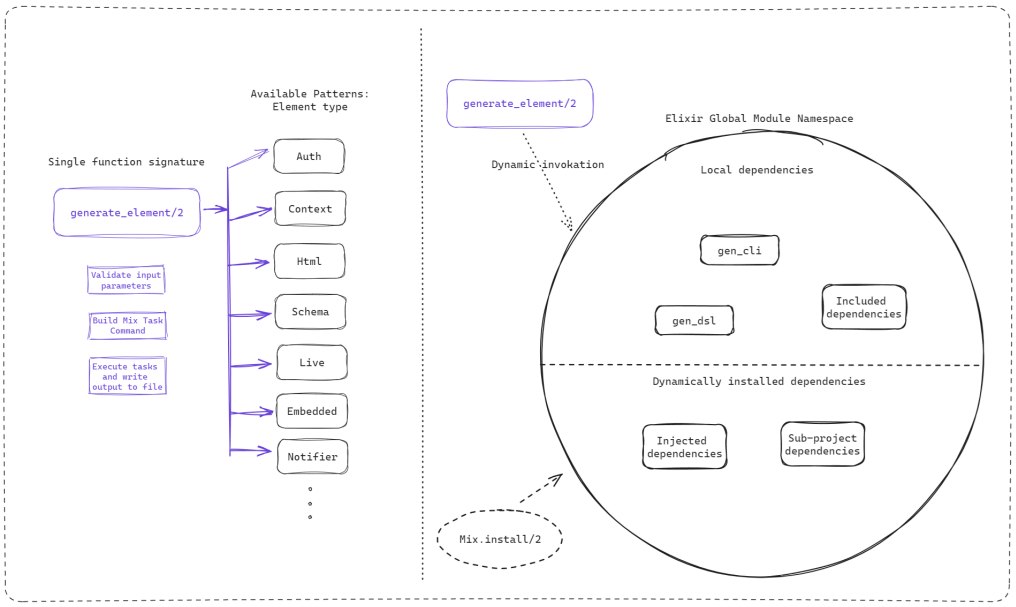

Figure 5.4: Generable element function matching comparison

5.3.2 Gen CLI

Implementing a plugin system for Gen DSL had upstream consequences in our toolchain. As we mentioned previously, every time Mix is invoked, a new VM is set up to, it loads globally installed packages, compiles the current directory's project if present, and loads it into memory. The abstraction for loading external dependencies during runtime used in Gen DSL hijacks this process using the `Mix.install/2` function, but since it can only be used once per VM, it prevents CLI from having access to it. This interfered with the implementation of Gen CLI at the time. When packaging it into an `mix archive` installable artifact, we lost the ability to bundle dependencies included in `mix.exs`. In an effort to overcome this

limitation we used `Mix.install/2` during its initialisation phase, a method we could no longer use.

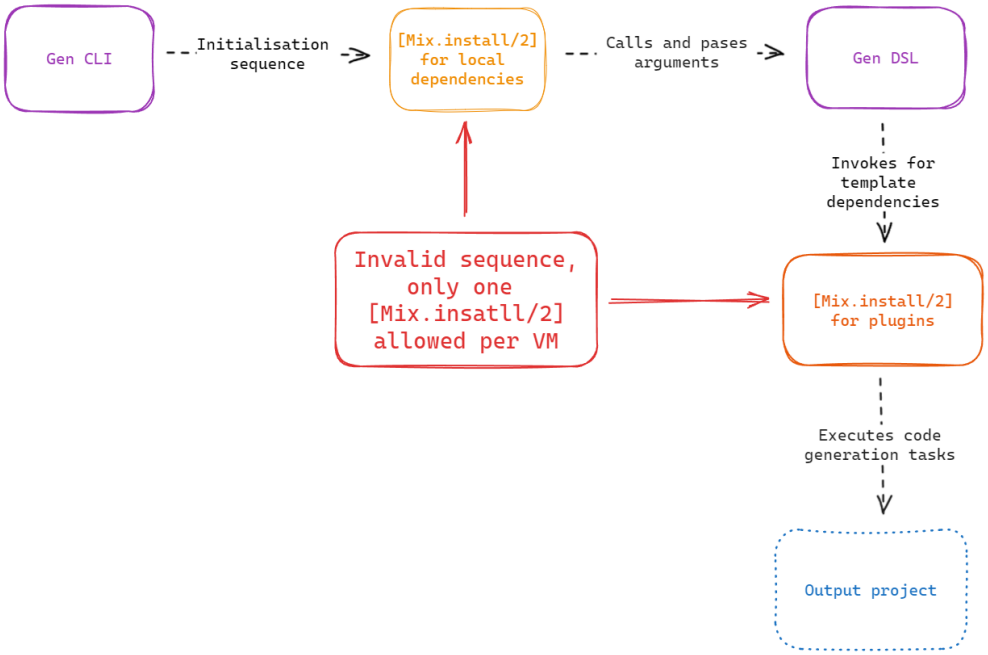

Figure 5.5: Invalid package install sequence

Our solution was to change our distribution approach from `mix archive` to `mix escript`. Escript is an alternative method for packaging applications provided by Mix specifically designed for CLI self-contained executables. With this method, we could bundle our application, all its dependencies, the Elixir runtime, and not have to worry about polluting Mix's dependency initialisation sequence. However, it came with its downside, mainly that it changed the invocation mechanism from a Mix task to an Erlang-interpreted binary, so users have to include it in their PATH environment variable to access it globally.

5.3.3 Gen Editor

During previous iterations, we established what technologies were suitable for the development of this package, but did not reach the same level of completeness as the rest of the toolchain. To accelerate the development process, we decided to replicate the implementation methodology employed by the core Livebook team. Consequently, we transitioned from vanilla JavaScript to Vue js, a client-side component-based reactive framework. Initially, we were against it since the BEAM community prefers to avoid JavaScript when possible, in favour of keeping state and rendering server-side. Ultimately, due to JavaScript's less terse and concise syntax, it ended up accounting for 64% of the package's lines of code.

With respect to the user interface, we started by designing how to add our Smart Cells to a notebook. Phoenix's code-generation tasks give access to 15 generable elements, but the cell variant selector API was only available for Livebook developers, as confirmed by a core member [33].

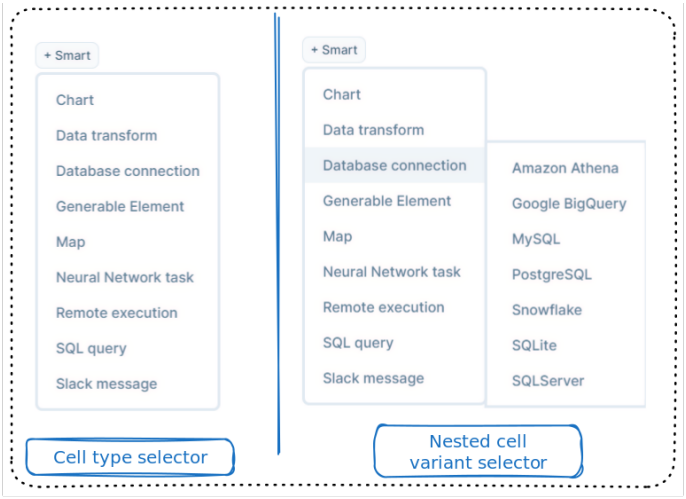

Figure 5.6: Livebook Smart Cell selector

To circumvent this limitation, we applied a common pattern found in other Kino packages. Instead of there being a cell for each generable element, we implemented a single cell that would render the appropriate form given a type selector. We also opted for only allowing a single element to be configured in each cell, as opposed to a cell containing multiple configurators. The main motivation was to encourage users to explain and document design decisions around each element utilising the rest of Livebook's cell types.

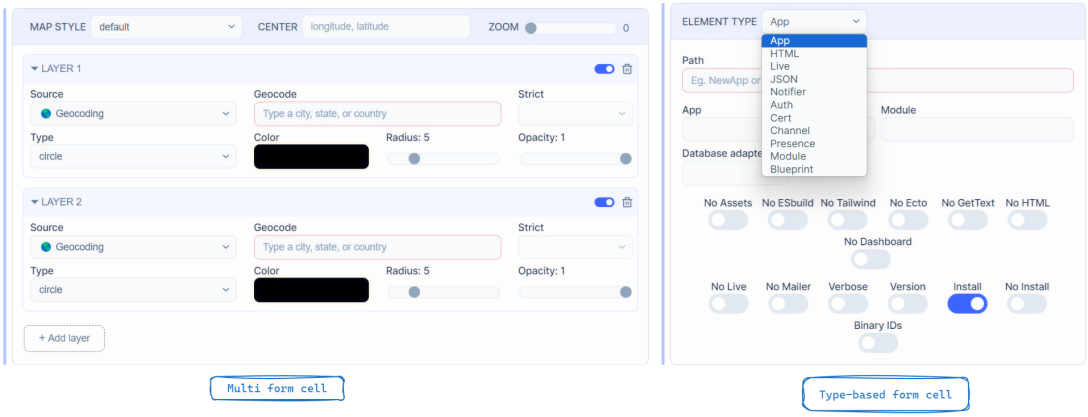

Figure 5.7: Cell UI style comparison

Earlier we noted there were functional dependencies between some elements 5.2.1, and some measures should be put into place to ensure correct project generation. Our solution was to allow users to mark elements as standalone or dependency, such that dependency element would not be generated but be available to other cells as arguments.

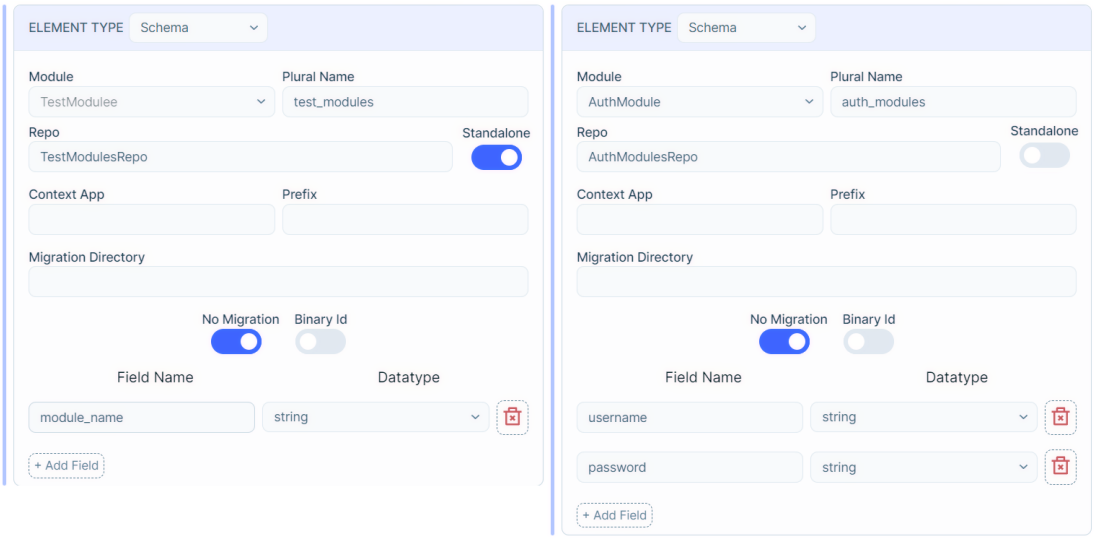

Figure 5.8: Standalone and dependency Schema elements

Gen DSL's data model does not allow for referencing elements recursively to prevent processing elements in a way that is too opinionated and push the burden to the template configuration workflow. Instead, in Gen Editor we add dependency element to the template as metadata, and during a processing step while building the output template, they get converted into the right parameters for those elements that depend on them.

Figure 5.9: Template processing step to solve element dependencies

Having overcome most technical challenges, we shifted our attention to implementing the quality-of-life enhancements outlined in the requirements:

- **Input validation**: Phoenix's tasks take some of their parameters as literals and since Elixir has a well-defined naming convention for its elements, it syntactically enforces snake_case for atoms and variables, period-separated PascalCase for modules and underscore-prefixed snake_case for metadata functions. We added client-side validation using regular expressions to check parameters adhered to the appropriate scheme.

- **Auto-complete**: to avoid errors when referencing other elements we originally proposed to provide argument auto-complete, however, in the end we decided a better option was to implement a dropdown selector with all the valid values.

- **In-editor documentation**: to prevent context switching, we wanted to provide quick and easy access to each Mix Tasks technical reference. We considered adding our custom explanation and specification to minimize the size taken by it and not disrupt the visual flow of the notebook. However, we ended up opting for a collapsible that embeds the official documentation for each element, facilitating up-to-date information and correctness.

- **Version control**: thanks to .livemd, Livebook's notebook file format, we got very good version control for our configurators without much effort. By implementing Kino's `to_attrs/1` callback for the parameters of each generable element, it automatically persisted configuration between sessions and wrote the state to file.
- **Collaborative editing**: in Livebook, the state of each Smart Cell is typically propagated to every user editing the document every time a cell is executed. That meant that intermediate changes to a cell were not visible to other users, manifesting in a poor collaborative experience. To solve it, we used Kino's `handle_info/2` and `broadcast_event/3` to propagate the state of any cell on every change, keeping any users in the notebook up to date.

Lastly, we incorporated the project generation workflow into Livebook as another type for the existing cell, as seen in 5.10. After processing the template, making the necessary changes, and generating elements, it allows users to download a well-formed `template.json` file as well as the project as a compressed archive.

Figure 5.10: Project generation workflow within Livebook

Chapter 6
Validation

In this first chapter, we will explain the testing approach taken for every package of the system.

6.1 Testing

Early in the project, we decided to limit testing of Gen Editor and Gen CLI to integration. Their tests are only meant to check if the interface between packages has been broken by any API changes. In contrast, Gen DSL received most of our attention, given that it contains the bulk of business logic and data modelling.

For Gen DSL we wanted to take inspiration from the Phoenix test suite, since it is a large and mature project with significant amounts of code generation. However, we discovered that Phoenix's testing for `mix phx.gen` commands is very minimal, and it boils down to checking if some of the expected files are present in the destination project directory. We wanted a more robust and exhaustive validation to encourage potential users, and set an example of testing procedure plugin developers can follow.

We ended up implementing a more involved approach, rooted in property-based testing [34], a testing philosophy for validating software programs in terms of their behaviour around a defined property. It does it by employing data generators following a specification (eg. all natural numbers or any alphanumeric string) to exhaustively check that a given function computes correct outputs with any value of its properties. In that spirit, we used the Stream Data library to define custom generators for each of Phoenix's generable elements. With this, we had access to arbitrary amounts of valid templates, but the main motivation was not to limit the assessment to correct execution, but also the validity of a project's contents. To achieve that, we implemented a property generation pipeline that dynamically builds property maps based on the specifics of a template. The main validations consist of checking modules for that element are present in the project, they are nested correctly within their context and

file structure, they contain the necessary functions to fulfil their expected interfaces, and such functions have the correct signature.

To perform such an exhaustive check over the generated project, we needed a powerful tool for code introspection. A choice seen in other Elixir projects was regular expressions to extract code fragments from source files, but we preferred a more semantic approach. Thus, we opt for parsing every single project file into an Abstract Syntax Tree, walking every node of the tree, and checking if it fulfils any property present in the property map. In the event it does, that property gets deleted from the map. As such, if the map is empty by the end of the validation, we consider that the project contains all the desired properties for a given template.

When tests are finished, we end up with three artifacts for every failed case. Its template, the generated project, and a file containing the properties it did not fulfil, making debugging significantly easier. The complete process can be seen in 6.1.

This testing pipeline provided considerable value to the project, uncovering significant misunderstandings on the constraints of each generable element, helping to define the data model more precisely, and providing a solid foundation for detecting regressions during development. Some notable examples are:

- Context parameters must be valid module names, not only valid atoms without prefix.
- Schema table names must be lowercase and plural.
- Modules represented as atoms must contain Elixir's global namespace prefix.
- Schemas used within other generable elements can contain fields, except when using them in combination with Auth elements.

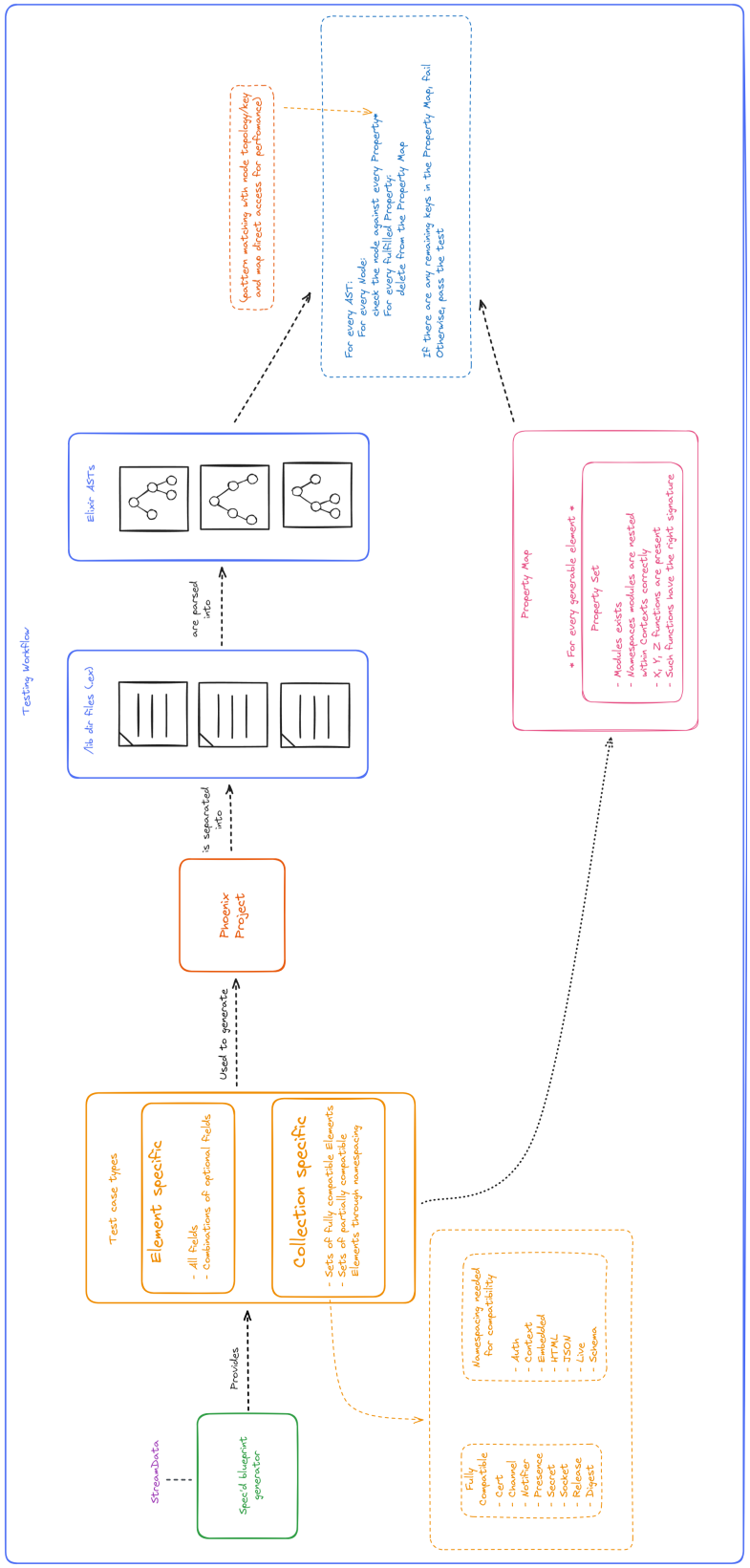

Figure 6.1: Testing pipeline for Gen DSL

Chapter 7

Extending the system

In this chapter we will describe the extensible features of the system, how they are implemented, and how to take advantage of them.

7.1 Extensibility mechanisms

As we have presented, the system consists of a series of tools that combine into a toolchain. The purpose of this design decision was to allow for incremental adoption, that is, for developers to be able to use the features they are interested in without having to adopt the entire system. Another aspect of this decision was to introduce the first of two methodologies for extensibility. By segmenting the process into building blocks, developers can add functionality to the system by implementing one of the packages by themselves, customising it to their liking.

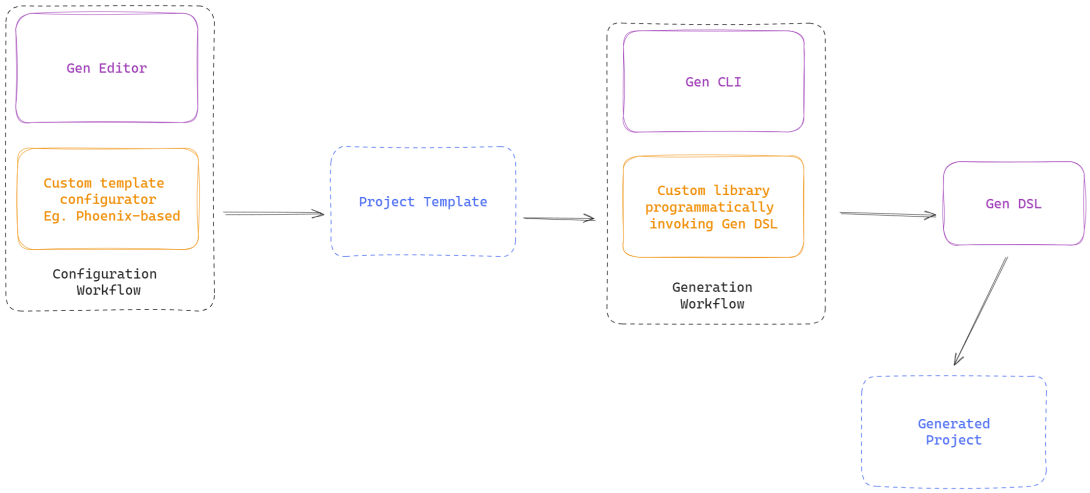

Figure 7.1: Example of alternative toolchain with third-party packages

The second method available for third-party developers provides a mechanism to directly add functionality to the system. By developing packages that follow defined interfaces, both Gen DSL and Gen Editor can incorporate ad-hoc functionality into the system, maintaining the benefits of the base implementation on top of the custom behaviour.

7.1.1 Extending Gen Editor

Those wishing to add custom template configurators can do it in two ways. By creating new graphical methods to configure existing generable elements or by adding configurators for new elements. To do so, developers must build a Smart Cell, complying with `Kino.SmartCell`'s requirements as seen in 5.3, that modifies the `blueprint` global variable accessible in the Livebook's session scope. They are allowed to add elements to existing keys(post-tasks, pre-tasks, generable_elements, dependencies, and metadata), but they must ensure that the added elements contain all the needed information for generation, as the final processing step mentioned here is not available to third-party cells. In addition, to maintain the current level of interactivity, they must notify all other cells using `broadcast_event/3` that there have been changes to `blueprint` and must recompute their state. To have access to these new cells alongside our system, users only have to install their own package using Livebook's dependencies setup in the same session.

7.1.2 Extending Gen DSL

As we mentioned previously, Gen DSL matches each generable element in a template to the corresponding execution method by selecting a function from the Elixir global module namespace with the supplied atom representation. As such, any package loaded into the runtime will be accessible to Gen DSL.

The method used to load dependencies into the system is not transparent to the user and has consequences to those adding functionality. As we described here, when Gen DSL is invoked, a VM is set up with its dependencies, those from an optional project in the current working directory, and the globally installed ones from `mix archive`. Afterwards, `Mix.install/2` is allowed to run once, and loads the dependencies described in the provided template. From here on, each generable element function must appropriately execute the accompanying Mix task, either on the same process or a new one. The dependencies available in each of them are different, so project bootstrapping tasks should be run from the main process, and tasks that depend on secondary packages present in the mix.exs of such generated project should be executed in a separate process. These methods are illustrated in 5.1

Developers can implement external packages in the form of plugins, which must follow a set of rules:

- **Define `to_task/1`**: a function that takes the attributes declared in the template for this element and returns a map with two keys. A `callback` containing reference to a first order function for code generation and an `arguments` key that contains the parameters needed for it.
- **Define `to_info/0`**: a function with no arguments that returns a description of the functionality of the plugin, or custom generable element.
- **OS command limitations**: all OS level commands used in the plugin, like Gen DSL does with `mkdir`, `rmdir`, and `tee`, must be POSIX compliant to maintain compatibility with most operating systems.

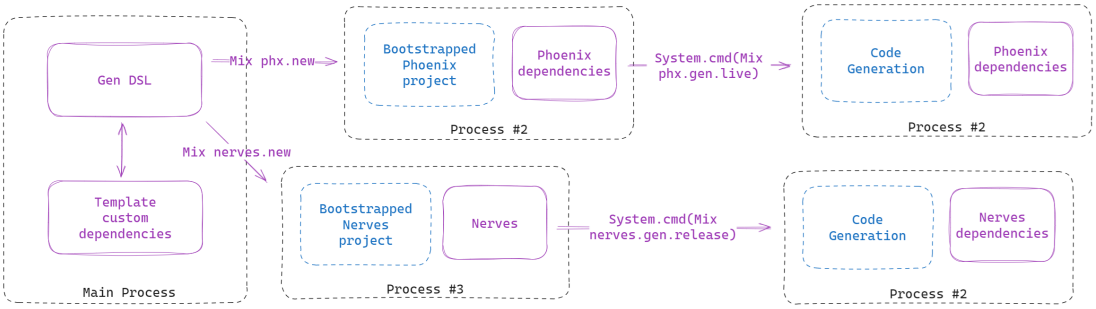

Figure 7.2: Example of process hierarchy during Gen DSL execution

The system to enforce this interface was implemented using Elixir Behaviours, a macro used to inject code into modules that include it. To encourage the community, we developed `mix gen_dsl_plugin.new`, a Mix Task to bootstrap a minimal plugin project.

Chapter 8
Conclusions

In this last chapter we will go over the final state of the project, the limitations in it, lessons we have learned along the way and what work could be done in the future.

8.1 State of the project

At this point in time, the project has met its initial objectives and adhered to the specified requirements. It provides an environment for users to design and document project templates based on Phoenix's `mix phx.gen` code generation Mix Tasks, with a much better developer experience achieved through collaborative editing, auto-complete, ease of versioning, syntax and command validation, and two separate workflows to avoid repetition and eliminate user error when replicating templates. The current implementation constitutes a strong foundation to build upon and has opened up opportunities to bring the same quality-of-life improvements to other libraries, like Nerves or the Ash Framework. However, the project has not realised our vision of a tool attractive enough to spark new interest for more advanced and composable Mix Tasks in the community, its current limitations around umbrella projects and supported Ecto datatypes could prevent teams from using it in production.

8.2 Limitations

The following shortcomings do not represent fundamental limitations in the context of the project, but rather trade-offs made in the spirit of keeping the scope of the system manageable:

- **Lack of support for Phoenix 1.7.11 and beyond**: due to the version freeze imposed during development to prevent chasing after other projects, we do not support all new generable element flags. However, we note that the effort required to update Gen DSL would be minimal.

- **Lack of support for Umbrella Project**: this alternative organisational structure for Elixir projects has significant differences and would require extensive architectural changes on how code generation tasks are executed. It is generally accepted in the community to not support them due to their lack of popularity, but going the extra mile to support them would pave the way for a more modular system and supporting Ash, a very popular framework.
- **Limited support for Ecto Schema types**: our generable element for Ecto schemas does not support Schema references, maps, arrays or recursive types, limiting users to the most basic uses.
- **Lack of concurrency support for code generation**: none of our generable elements can be generated in parallel, as we do not handle interactions with the file system, and severely limit the speed at which projects can be generated.
- **Gen DSL's VMs management is a leaky abstraction**: which forces any other package that interacts with it to take it into account. This is a major source of friction, but a redesign of the package's pipeline would not be needed to limit its reach.

8.3 Lessons learned

Throughout the development of this project we have acquired a more intimate knowledge of the technologies involved, these are opinions we have formed about them:

- Livebook offers a very good developer experience when developing integrations, and as more of them are released it keeps improving as a prototyping and documentation tool that offers a very solid foundation on which to build our own applications, especially with the deploy functionality recently added.
- Elixir is not suitable for CLI applications due to limitations when bundling applications. Building a self-contained executable is not supported by the official tooling, making Erlang or Elixir mandatory external dependencies.
- Mix Tasks are not prepared to be invoked programmatically. They require hacks and workarounds when used outside the same directory the main program is invoked. Capturing its output requires the use of OS level utilities like `cat` or `tee` to capture their output and there is no idiomatic way to handle interactive prompts.
- Terminal commands executed from the Erlang VM are inherently flaky due to not returning EOF for binary outputs in `stdout`, hanging the terminal output. This has been reported and marked as won't fix by its maintainers [35].

If we had the opportunity to start the project from scratch, we would offload any business logic associated with Mix Tasks to Igniter, a newly released tool for composing Mix Tasks and adding project dependencies, created by the maintainers of Ash.

8.4 Further work

Our project constitutes a solid and modular foundation that would benefit from additional work. Thanks to the open-source approach, everything needed for members of the community to start contributing has been set up using the most popular channels and publicly documented. Besides correcting the aforementioned limitations, the following features would have the biggest impact on the usefulness of the system:

- Reimplement Mix Tasks invocation and dependency management using Igniter, since despite being a very new tool, it is rising in popularity and it provides a remarkable set of features, as well as being extensible by nature.
- Add a new generable element to Gen DSL that allows users to compose third-party commands without implementing their own plugin, reducing the barrier of entry of custom elements.
- Add DBML compatibility in Gen Editor for generating Schemas, to be able to make use of an existing ecosystem of tools, including drag-and-drop editors such as DBDiagram.io.
- Implement automatic diagram generation to aid in documenting generated projects using tools like xref, Ecto ERD, PUML generator, dep_viz and Kino.Process.

Appendices

Appendix A

Data model

This chapter provides a complete implementation reference for Gen DSL's data model.

- **App**: Creates a new Phoenix project.

```
    defmodule GenDSL.Model.App do
      use Ecto.Schema
      import Ecto.Changeset

      schema "App" do
        field(:path, :string)
        field(:umbrella, :boolean)
        field(:app, :string)
        field(:module, :string)
        field(:database, Ecto.Enum, values: [:postgres, :mysql,
    :mssql, :sqlite3])
        field(:no_assets, :boolean)
        field(:no_esbuild, :boolean)
        field(:no_tailwind, :boolean)
        field(:no_dashboard, :boolean)
        field(:no_ecto, :boolean)
        field(:no_gettext, :boolean)
        field(:no_html, :boolean)
        field(:no_live, :boolean)
        field(:no_mailer, :boolean)
        field(:binary_id, :boolean)
        field(:verbose, :boolean)
        field(:install, :boolean)
        field(:no_install, :boolean)
        field(:log_filepath, :string, default: "INSTRUCTIONS.md")
        field(:command, :string, default: "new")
      end

      @required_fields ~w[path]a
```

```
    @optional_fields ~w[umbrella app module database no_assets
no_esbuild no_tailwind no_dashboard no_ecto no_gettext no_html
no_live no_mailer binary_id verbose install no_install]a

    @flags ~w[umbrella no_assets no_esbuild no_tailwind
no_dashboard no_ecto no_gettext no_html no_live no_mailer
binary_id verbose install no_install]a
    @named_arguments ~w[app module database]a
    @positional_arguments ~w[path]a

    def changeset(params \\ %{}) do
      %__MODULE__{}
      |> cast(params, @required_fields ++ @optional_fields,
required: false)
      |> validate_required(@required_fields)
    end
  end

```

- **Auth**: Generates authentication logic and related views for a resource.

```
  defmodule GenDSL.Model.Auth do
        use Ecto.Schema
        import Ecto.Changeset

        schema "Auth" do
          field(:context, :string)
          field(:web, :string)
          field(:hashing_lib, Ecto.Enum, values: [bcrypt:
"bcrypt", pbkdf2: "pbkdf2", argon2: "argon2"])
          field(:no_live, :boolean, default: false)
          field(:live, :boolean, default: true)
          field(:merge_with_existing_context, :boolean, default:
true)
          embeds_one(:schema, GenDSL.Model.Schema)
          field(:path, :string)
          field(:log_filepath, :string, default:
"INSTRUCTIONS.md")
          field(:command, :string, default: "auth")
        end

        @required_fields ~w[context path]a
        @optional_fields ~w[web hashing_lib no_live live
merge_with_existing_context]a
        @remainder_fields ~w[]a

```

```
        @flags ~w[no_live live merge_with_existing_context]a
        @named_arguments ~w[web hashing_lib]a
        @positional_arguments ~w[context]a

        def changeset(params \\ %{}) do
          %__MODULE__{}
          |> cast(params, @required_fields ++ @optional_fields ++ @remainder_fields, required: false)
          |> cast_embed(:schema, required: true, with: &GenDSL.Model.Schema.embedded_changeset/2)
          |> validate_required(@required_fields)
        end
      end

```

- **Cert**: Generates a self-signed certificate for HTTPS testing.

```
defmodule GenDSL.Model.Cert do
  use Ecto.Schema
  import Ecto.Changeset

  schema "Cert" do
    field(:app, :string)
    field(:domain, :string)
    field(:url, :string)
    field(:output, :string)
    field(:name, :string)

    field(:path, :string)
    field(:log_filepath, :string, default: "INSTRUCTIONS.md")
    field(:command, :string, default: "cert")
  end

  @required_fields ~w[path]a
  @optional_fields ~w[app domain url output name]a

  @flags ~w[]a
  @named_arguments ~w[output name]a
  @positional_arguments ~w[app domain url]a

  def changeset(params \\ %{}) do
    %__MODULE__{}
    |> cast(params, @required_fields ++ @optional_fields, required: false)
    |> validate_required(@required_fields)
  end
```

```
    end

```

- **Channel**: Generates a Phoenix channel for soft real-time data.

```
    defmodule GenDSL.Model.Channel do
          use Ecto.Schema
          import Ecto.Changeset

          schema "Channel" do
            field(:module, :string)

            field(:path, :string)
            field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
            field(:command, :string, default: "channel")
          end

          @required_fields ~w[module path]a
          @optional_fields ~w[]a

          @flags ~w[]a
          @named_arguments ~w[]a
          @positional_arguments ~w[module]a

          def changeset(params \\ %{}) do
            %__MODULE__{}
            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
            |> validate_required(@required_fields)
          end
      end

```

- **Context**: Generates a context with functions around an Ecto schema.

```
        defmodule GenDSL.Model.Context do
          use Ecto.Schema
          import Ecto.Changeset

          schema "Context" do
            field(:context, :string)
            field(:no_schema, :boolean, default: false)
            field(:merge_with_existing_context, :boolean, default:
    true)
            field(:no_merge_with_existing_context, :boolean,
    default: false)
```

```
10
11            field(:path, :string)
12            field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
13            embeds_one(:schema, GenDSL.Model.Schema)
14
15            field(:command, :string, default: "context")
16          end
17
18          @required_fields ~w[context path]a
19          @optional_fields ~w[no_schema merge_with_existing_context
    no_merge_with_existing_context]a
20          @remainder_fields ~w[]a
21
22          @flags ~w[no_schema merge_with_existing_context
    no_merge_with_existing_context]a
23          @named_arguments ~w[]a
24          @positional_arguments ~w[context]a
25
26          def changeset(params \\ %{}) do
27            %__MODULE__{}
28            |> cast(params, @required_fields ++ @optional_fields ++
    @remainder_fields, required: false)
29            |> cast_embed(:schema, required: false, with:
    &GenDSL.Model.Schema.embedded_changeset/2)
30            |> validate_required(@required_fields)
31          end
32        end
33
```

- **Embedded**: Generates an embedded Ecto schema for casting/validating data outside the DB.

```
1         defmodule GenDSL.Model.Embedded do
2           use Ecto.Schema
3           import Ecto.Changeset
4
5           schema ":Embedded" do
6             field(:module, :string)
7             field(:path, :string)
8             field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
9             field(:command, :string, default: "embedded")
10
11            embeds_many(:fields, GenDSL.Model.SchemaField)
12          end
```

```
13
14          @required_fields ~w[module path]a
15          @optional_fields ~w[]a
16
17          @flags ~w[]a
18          @named_arguments ~w[]a
19          @positional_arguments ~w[module]a
20
21          def changeset(params \\ %{}) do
22            %__MODULE__{}
23            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
24            |> cast_embed(:fields, required: false)
25            |> validate_required(@required_fields)
26          end
27
28          def embedded_changeset(_parent, params \\ %{}) do
29            %__MODULE__{}
30            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
31            |> cast_embed(:fields, required: false)
32            |> validate_required(@required_fields)
33          end
34        end
35
```

- **Html**: Generates controller with view, templates, schema and context for an HTML resource.

```
1         defmodule GenDSL.Model.Html do
2           use Ecto.Schema
3           import Ecto.Changeset
4
5           schema "Html" do
6             field(:context, :string)
7             field(:web, :string)
8             field(:no_context, :boolean, default: false)
9             field(:no_schema, :boolean, default: false)
10            field(:merge_with_existing_context, :boolean, default:
    true)
11            field(:context_app, :string)
12            embeds_one(:schema, GenDSL.Model.Schema)
13
14            field(:path, :string)
15            field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
```

```
16            field(:command, :string, default: "html")
17          end
18
19          @required_fields ~w[context path]a
20          @optional_fields ~w[web context_app no_context no_schema
    merge_with_existing_context]a
21          @remainder_fields ~w[]a
22
23          @flags ~w[no_context no_schema
    merge_with_existing_context]a
24          @named_arguments ~w[web context_app]a
25          @positional_arguments ~w[context]a
26
27          def changeset(params \\ %{}) do
28            %__MODULE__{}
29            |> cast(params, @required_fields ++ @optional_fields ++
    @remainder_fields, required: false)
30            |> cast_embed(:schema, required: false, with:
    &GenDSL.Model.Schema.embedded_changeset/2)
31            |> validate_required(@required_fields)
32          end
33        end
34
```

- **Json**: Generates controller, JSON view, and context for a JSON resource, akin to a REST API.

```
1       defmodule GenDSL.Model.Json do
2         use Ecto.Schema
3         import Ecto.Changeset
4
5         schema "Json" do
6           field(:context, :string)
7           field(:web, :string)
8           field(:context_app, :string)
9           field(:no_context, :boolean, default: false)
10          field(:no_schema, :boolean, default: false)
11          field(:merge_with_existing_context, :boolean, default:
    true)
12
13          embeds_one(:schema, GenDSL.Model.Schema)
14
15          field(:path, :string)
16          field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
17          field(:command, :string, default: "json")
```

```
18          end
19
20          @required_fields ~w[context path]a
21          @optional_fields ~w[web context_app
    merge_with_existing_context no_context no_schema]a
22          @remainder_fields ~w[]a
23
24          @flags ~w[merge_with_existing_context no_context
    no_schema]a
25          @named_arguments ~w[web context_app]a
26          @positional_arguments ~w[context]a
27
28          def changeset(params \\ %{}) do
29            %__MODULE__{}
30            |> cast(params, @required_fields ++ @optional_fields ++
    @remainder_fields, required: false)
31            |> cast_embed(:schema, required: false, with:
    &GenDSL.Model.Schema.embedded_changeset/2)
32            |> validate_required(@required_fields)
33          end
34        end
35
```

- **Live**: Generates LiveView, templates, and context for a resource with soft real-time communication with the server.

```
1       defmodule GenDSL.Model.Live do
2         use Ecto.Schema
3         import Ecto.Changeset
4
5         schema "Live" do
6           field(:context, :string)
7           field(:web, :string)
8           field(:no_context, :boolean)
9           field(:no_schema, :boolean)
10          field(:merge_with_existing_context, :boolean, default:
    true)
11          field(:context_app, :string)
12
13          embeds_one(:schema, GenDSL.Model.Schema)
14
15          field(:path, :string)
16          field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
17          field(:command, :string, default: "live")
18        end
```

```
19
20          @required_fields ~w[context path]a
21          @optional_fields ~w[web no_context no_schema context_app
    merge_with_existing_context]a
22          @remainder_fields ~w[]a
23
24          @flags ~w[no_context no_schema
    merge_with_existing_context]a
25          @named_arguments ~w[web context_app]a
26          @positional_arguments ~w[context]a
27
28          def changeset(params \\ %{}) do
29            %__MODULE__{}
30            |> cast(params, @required_fields ++ @optional_fields ++
    @remainder_fields, required: false)
31            |> cast_embed(:schema, required: false, with:
    &GenDSL.Model.Schema.embedded_changeset/2)
32            |> validate_required(@required_fields)
33          end
34        end
35
```

- **Notifier**: Generates a notifier that delivers emails by default.

```
1         defmodule GenDSL.Model.Notifier do
2           use Ecto.Schema
3           import Ecto.Changeset
4
5           schema "Notifier" do
6             field(:context, :string)
7             field(:module, :string)
8             field(:message_names, {:array, :string})
9             field(:merge_with_existing_context, :boolean, default:
    true)
10            field(:context_app, :string)
11
12            field(:path, :string)
13            field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
14            field(:command, :string, default: "notifier")
15          end
16
17          @required_fields ~w[context module message_names path]a
18          @optional_fields ~w[context_app
    merge_with_existing_context]a
19
```

```
          @flags ~w[merge_with_existing_context]a
          @named_arguments ~w[context_app]a
          @positional_arguments ~w[context module message_names]a

          def changeset(params \\ %{}) do
            %__MODULE__{}
            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
            |> validate_required(@required_fields)
          end
        end

```

- **Presence**: Generates a Presence tracker for users in a given route.

```
        defmodule GenDSL.Model.Presence do
          use Ecto.Schema
          import Ecto.Changeset

          schema "Presence" do
            field(:module, :string, default: "Presence")

            field(:path, :string)
            field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
            field(:command, :string, default: "presence")
          end

          @required_fields ~w[path]a
          @optional_fields ~w[module]a

          @flags ~w[]a
          @named_arguments ~w[]a
          @positional_arguments ~w[module]a

          def changeset(params \\ %{}) do
            %__MODULE__{}
            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
            |> validate_required(@required_fields)
          end
        end

```

- **Release**: Generates release files and optional Dockerfile for release-based deployments.

```
      defmodule GenDSL.Model.Release do
        use Ecto.Schema
        import Ecto.Changeset

        schema "Release" do
          field(:docker, :boolean)
          field(:no_ecto, :boolean)
          field(:ecto, :boolean)

          field(:path, :string)
          field(:log_filepath, :string, default:
"INSTRUCTIONS.md")
          field(:command, :string, default: "release")
        end

        @required_fields ~w[path]a
        @optional_fields ~w[docker no_ecto ecto]a

        @flags ~w[docker no_ecto ecto]a
        @named_arguments ~w[]a
        @positional_arguments ~w[]a

        def changeset(params \\ %{}) do
          %__MODULE__{}
          |> cast(params, @required_fields ++ @optional_fields,
required: false)
          |> validate_required(@required_fields)
        end
      end

```

- **Schema and SchemaField**: Generates an Ecto schema and migration.

```
      defmodule GenDSL.Model.Schema do
        use Ecto.Schema
        import Ecto.Changeset

        schema ":Schema" do
          field(:module, :string)
          field(:plural, :string)
          field(:table, :string)
          field(:repo, :string)
          field(:migration_dir, :string)
          field(:prefix, :string)
          field(:no_migration, :boolean)
          field(:binary_id, :boolean)
```

```
14            field(:context_app, :string)
15
16            field(:path, :string)
17            field(:log_filepath, :string, default:
    "INSTRUCTIONS.md")
18            field(:command, :string, default: "schema")
19
20            embeds_many(:fields, GenDSL.Model.SchemaField)
21          end
22
23          @required_fields ~w[module plural path]a
24          @optional_fields ~w[no_migration table binary_id repo
    migration_dir prefix context_app]a
25
26          @flags ~w[no_migration binary_id]a
27          @named_arguments ~w[table repo migration_dir prefix
    context_app]a
28          @positional_arguments ~w[module plural]a
29
30          def changeset(params \\ %{}) do
31            %__MODULE__{}
32            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
33            |> cast_embed(:fields, required: false)
34            |> validate_required(@required_fields)
35          end
36
37          def embedded_changeset(_parent, params \\ %{}) do
38            %__MODULE__{}
39            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
40            |> cast_embed(:fields, required: false)
41            |> validate_required(@required_fields)
42          end
43          defmodule GenDSL.Model.SchemaField do
44              use Ecto.Schema
45              import Ecto.Changeset
46
47              schema ":SchemaField" do
48                field(:field_name, :string)
49                field(:datatype, Ecto.Enum,
50                  values: [
51                    :id,
52                    :binary_id,
53                    :integer,
54                    :float,
```

```
                    :boolean,
                    :string,
                    :binary,
                    :map,
                    :decimal,
                    :date,
                    :time,
                    :time_usec,
                    :naive_datetime,
                    :naive_datetime_usec,
                    :utc_datetime,
                    :utc_datetime_usec
                  ]
                )
              end

              @required_fields ~w[field_name datatype]a
              @optional_fields ~w[]a

              # @flags ~w[]a
              # @named_parameters ~w[field_name datatype]a

              def changeset(_, params \\ %{}) do
                %__MODULE__{}
                |> cast(params, @required_fields ++
@optional_fields, required: false)
                |> validate_required(@required_fields)
              end
            end

```

- **Secret**: Generates a secret and prints it to the terminal.

```
      defmodule GenDSL.Model.Secret do
        use Ecto.Schema
        import Ecto.Changeset

        schema "Secret" do
          field(:length, :integer, default: 32)

          field(:path, :string)
          field(:log_filepath, :string, default:
"INSTRUCTIONS.md")
          field(:command, :string, default: "secret")
        end

```

```
13          @required_fields ~w[path]a
14          @optional_fields ~w[length]a
15
16          @flags ~w[]a
17          @named_arguments ~w[]a
18          @positional_arguments ~w[length]a
19
20          def changeset(params \\ %{}) do
21            %__MODULE__{}
22            |> cast(params, @required_fields ++ @optional_fields,
    required: false)
23            |> validate_required(@required_fields)
24          end
25        end
26
```

- **Socket**: Generates a Phoenix socket handler.

```
1   defmodule GenDSL.Model.Socket do
2     use Ecto.Schema
3     import Ecto.Changeset
4
5     schema "Socket" do
6       field(:module, :string)
7
8       field(:path, :string)
9       field(:log_filepath, :string, default: "INSTRUCTIONS.md")
10      field(:command, :string, default: "socket")
11    end
12
13    @required_fields ~w[module path]a
14    @optional_fields ~w[]a
15
16    @flags ~w[]a
17    @named_arguments ~w[]a
18    @positional_arguments ~w[module]a
19
20    def changeset(params \\ %{}) do
21      %__MODULE__{}
22      |> cast(params, @required_fields ++ @optional_fields,
    required: false)
23      |> validate_required(@required_fields)
24    end
25  end
26
```

List of Acronyms

BEAM/ErlangVM Bogdan Erlang Abstract Machine. 2

CLI Command Line-Interface. 3

CRUD Create Read Update Delete. 20

DBML DataBase Markup Language. 30, 49

JSON JavaScript Object Notation. 25, 29

MVC Model View Controller. 3

PR Pull Request. 8–10

TOML Tom's Obvious Minimal Language. 29

YAML Yet Another Markup Language. 29

Glossary

Abstract Syntax Tree A tree representation of the abstract syntactic structure of text written in a formal language. Each node of the tree denotes a construct occurring in the source code. 42

Ash Framework A declarative, extensible framework for building Elixir applications. 4

atom Constant whose value is its own name. Commonly used for identifying global and unique elements. 45

CSS Cascading Style Sheets is a style sheet language used for specifying the presentation and styling of a document such as Html or XML. 30

DBDiagram.io A free, simple tool to draw ER diagrams by just writing code. 30, 49

dep_viz A visual tool to help developers understand Elixir recompilation in their projects. 49

Discord Instant messaging and VoIP social platform which allows communication through voice calls, video calls, text messaging, and media and files. 7

Ecto ERD Mix task for generating Entity Relationship Diagram from Ecto schemas available in your project. 49

ECTS European Credit Transfer and Accumulation System. 11

Elixirforum Official forum for Elixir programming language enthusiasts. 7

escript Executable that can be invoked from the command line. An escript can run on any machine that has Erlang/OTP installed and by default does not require Elixir to be installed, as Elixir is embedded as part of the escript. 24

Exconstructor Elixir library that makes it easy to instantiate structs from external data, such as that emitted by a JSON parser. 31

GitHub Developer platform that allows developers to create, store, manage and share their code. 11

Hex The package manager for the Erlang ecosystem, available through Mix and Rebar3. 11, 32

Igniter Code generation and project patching framework. 49

Java A high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let programmers write once, run anywhere. 32

JavaScript Programming language and core technology of the Web, alongside HTML and CSS, commonly used to adding interactive elements to the web. 32

Jupyter Notebooks Web-based interactive development environment for notebooks, code, and data. Its flexible interface allows users to configure and arrange workflows in data science, scientific computing, computational journalism, and machine learning. 4

Kino.Process Module that contains kinos for generating visualizations to help introspect your running processes. 49

macro Compile-time constructs that are invoked with Elixir's AST as input and a superset of Elixir's AST as output. 31

Matplotlib A comprehensive library for creating static, animated, and interactive visualizations in Python. 4

microservices Architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. 26

Nerves Project Open-source platform that combines the rock-solid BEAM virtual machine and Elixir ecosystem to easily build and deploy production embedded systems. 4

NimbleParsec A simple and fast library for text-based parser combinators. 29

PascalCase Naming convention in which there are no spaces and the first character of every word is written in uppercase. 39

Poison An incredibly fast, pure Elixir JSON library. 25, 29

POSIX The Portable Operating System Interface is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. 46

PUML generator Sequence diagrams generator for Elixir Phoenix projects. 49

Slack Cloud-based team communication platform developed by Slack Technologies. 7

snake_case naming convention in which each space is replaced with an underscore character, and words are written in lowercase. 39

Stream Data Elixir library for data generation and property-based testing for Elixir. 41

Vue js Approachable, performant and versatile framework for building Single Page Applications. 36

xref Prints cross reference information between modules. 49

Bibliography

[1] T. N. Theis and H.-S. P. Wong, "The End of Moore's Law: A New Beginning for Information Technology," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, Mar. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7878935/

[2] Honeypot, "Elixir: The Documentary," Jul. 2018. [Online]. Available: https://www.youtube.com/watch?v=lxYFOM3UJzo

[3] G. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998. [Online]. Available: http://ieeexplore.ieee.org/document/658762/

[4] H. Sutter and J. Larus, "Software and the Concurrency Revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry." *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005. [Online]. Available: https://dl.acm.org/doi/10.1145/1095408.1095421

[5] T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere, "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects," in *2013 IEEE 37th Annual Computer Software and Applications Conference*. Kyoto, Japan: IEEE, Jul. 2013, pp. 303–312. [Online]. Available: https://ieeexplore.ieee.org/document/6649842

[6] E. S. team, 2020. [Online]. Available: https://www.erlang-solutions.com/blog/optimising-for-concurrency-comparing-and-contrasting-the-beam-and-jvm-virtual-machines/

[7] E. AB, "Erlang - Introduction," 2024. [Online]. Available: https://www.erlang.org/doc/system_architecture_intro/sys_arch_intro.html#id58791

[8] *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM Digital Library, 2013, oCLC: 1181192242.

[9] D. H. Hansson, "The one person framework," 2021. [Online]. Available: https://world.hey.com/dhh/the-one-person-framework-711e6318

[10] "Livebook documentation," [Accessed 4-February-2024]. [Online]. Available: https://hexdocs.pm/livebook/readme.html

[11] Y. Su, "Livebook-driven development," Jun. 2021, [Accessed 10-February-2024]. [Online]. Available: https://goofansu.medium.com/livebook-driven-development-50f82e66fbff

[12] A. Lancaster, "Livebook for Elixir: Just what the docs ordered," May 2022, [Accessed 10-February-2024]. [Online]. Available: https://blog.appsignal.com/2022/05/24/livebook-for-elixir-just-what-the-docs-ordered.html

[13] J. Valim, "Deploy notebooks as apps," Apr. 2023, [Accessed 10-February-2024]. [Online]. Available: https://news.livebook.dev/deploy-notebooks-as-apps-quality-of-life-upgrades---launch-week-1---day-1-2OTEWI

[14] J. Bukanek, "Model-View-Controller Pattern," in *Learn Objective-C for Java Developers*. Berkeley, CA: Apress, 2009, pp. 353–402. [Online]. Available: http://link.springer.com/10.1007/978-1-4302-2370-2_20

[15] A. Conrad, "Chris McCord on Phoenix's LiveView Functionality," *IEEE Software*, vol. 37, no. 3, pp. 98–100, May 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9068368/

[16] Rennie, Gary, "The road to 2 million Websocket Connections in Phoenix," Nov. 2015, [Accessed 4-February-2024]. [Online]. Available: https://www.phoenixframework.org/blog/the-road-to-2-million-websocket-connections

[17] "2022 Developer Survey," Nov. 2022, [Accessed 5-February-2024]. [Online]. Available: https://survey.stackoverflow.co/2022

[18] "2023 Developer Survey," Nov. 2023, [Accessed 5-February-2024]. [Online]. Available: https://survey.stackoverflow.co/2023

[19] McCord, Chris, "Phoenix 1.0 – the framework for the modern web just landed," [Accessed 6-February-2024]. [Online]. Available: https://www.phoenixframework.org/blog/phoenix-10-the-framework-for-the-modern-web-just-landed

[20] "Erlang dialyzer documentation," [Accessed 16-June-2024]. [Online]. Available: https://www.erlang.org/docs/23/man/dialyzer.html

[21] "Phx gen documentation," [Accessed 6-February-2024]. [Online]. Available: https://hexdocs.pm/phoenix/Mix.Tasks.Phx.Gen.html

[22] "Mix tasks documentation," [Accessed 6-February-2024]. [Online]. Available: https://hexdocs.pm/mix/1.16.1/Mix.Task.html

[23] H. Sampath, A. Merrick, and A. Macvean, "Accessibility of Command Line Interfaces," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Yokohama Japan: ACM, May 2021, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3411764.3445544

[24] B. Wichmann, A. Canning, D. Marsh, D. Clutterbuck, L. Winsborrow, and N. Ward, "Industrial perspective on static analysis," *Software Engineering Journal*, vol. 10, no. 2, p. 69, 1995. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/sej.1995.0010

[25] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo, "Beyond source code: The importance of other artifacts in software development (a case study)," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1233–1248, Sep. 2006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121206000732

[26] L. San Román, "A deep dive into the elixir ast," Apr. 2021, [Accessed 11-February-2024]. [Online]. Available: https://dorgan.ar/posts/2021/04/the_elixir_ast/

[27] G. Open Source Community, "Open source guidelins," [Accessed 8-September-2023]. [Online]. Available: https://opensource.guide/

[28] "Gitflow workflow," [Accessed 16-June-2024]. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[29] "Github flow," [Accessed 16-June-2024]. [Online]. Available: https://docs.github.com/en/get-started/using-github/github-flow

[30] GitHub, "Making your life easier as an open source maintainer, from documenting processes to leveraging your community." 2024, last accessed September 12, 2024. [Online]. Available: https://opensource.guide/

[31] M. de Trabajo y Economía Social, "Resolución de 27 de febrero de 2023, de la dirección general de trabajo, por la que se registra y publica el xx convenio colectivo nacional de empresas de ingeniería; oficinas de estudios técnicos; inspección, supervisión y control técnico y de calidad." [Accessed 16-June-2024]. [Online]. Available: https://www.boe.es/eli/es/res/2023/02/27/(6)

[32] B. Gustavsson, "Erlang 27 changelog," 2024. [Online]. Available: https://www.erlang.org/blog/highlights-otp-27/

[33] J. Klosko, "Limitations of livebook's cell registration api," 2023. [Online]. Available: https://elixirforum.com/t/how-to-register-a-livebook-smart-cell-with-sub-types/58835/2

[34] L. M. C. Souto, "Making property based testing easier," 2016. [Online]. Available: https://www.youtube.com/watch?v=1n650p3CQgo

[35] vans163, "os:cmd not returning all of stdout when stdout output is binary," 2016. [Online]. Available: https://github.com/erlang/otp/issues/3372