BUILDING WEB APPLICATIONS WITH SERVANT IN HASKELL

A PRACTICAL GUIDE TO TYPE-SAFE APIS FROM DESIGN TO DEPLOYMENT

BUILD A COMPLETE, VERSIONED REST API
WITH REAL-WORLD PATTERNS AND BEST PRACTICES

JOHANNES HOFFMAN

Building Web Applications with Servant in Haskell

A Practical Guide to Type-Safe APIs from Design to Deployment

Steve Publications

This book is available at
https://leanpub.com/buildingwebapplicationswithservantinhaskell

This version was published on 2026-08-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Practical Guide to Type-Safe APIs from Design to Deployment

This book teaches you how to build production-quality web applications and REST APIs using the Servant framework in Haskell. Starting from foundational Haskell concepts, it guides you through type-level API design, database integration, authentication, testing, and deployment. By the end, you will have built a complete versioned REST API with PostgreSQL persistence, JWT authentication, structured error handling, automated tests, and Docker configuration that demonstrates all the patterns used in real-world Servant applications.

Introduction

Imagine defining your entire web API as a single Haskell type. From that one declaration, the compiler guarantees that your server handlers match every endpoint correctly. It generates type-safe client functions you can use from other Haskell programs. It produces machine-readable documentation that stays synchronized with your code because it is derived from the same source. This is not science fiction or an aspirational roadmap. This is what Servant does today, and it changes how you think about building web APIs.

Most web frameworks treat API design as a runtime concern. You write route handlers with string-based paths, document endpoints manually in some external tool, and hope your client code stays in sync when you change the server. A typo in a URL pattern becomes a 404 error that only surfaces when someone actually requests that path. Rename a JSON field on the server and every consuming client breaks at runtime. These problems are not theoretical edge cases. They are daily realities for teams maintaining large API codebases.

Servant takes a fundamentally different approach by leveraging Haskell's powerful type system to move as much of the web development process to compile time as possible. The core idea is simple but profound: describe your API as a type, then interpret that type in multiple ways. One interpretation gives you a routing table for your server. Another gives you client functions. Another gives you documentation. Because all three come from the same type-level description, they cannot drift apart without the compiler telling you about it.

This book assumes you understand basic programming concepts like variables, functions, data structures, and control flow. You do not need prior Haskell experience, but you should be comfortable with the idea that a language's type system can express more than just "this value is an integer" or "this is a list of strings." If you have used TypeScript, Rust, Scala, or any other language with non-trivial types, you will find many concepts familiar.

The journey ahead is structured as a progressive learning path. We begin by understanding the Haskell web ecosystem and what makes Servant distinctive among available frameworks. Then we set up your development environment

so you can follow along with every example. Since Servant relies heavily on advanced Haskell features like type-level programming, monads, and monad transformers, we take time to build those foundations before diving into API design.

Once the foundations are in place, you will define your first API routes and implement handlers that serve real data. We explore every combinator Servant provides for describing endpoints: path parameters, query strings, request bodies, headers, status codes, and content types. You will learn how JSON serialization works with Aeson, how to structure larger APIs by composing smaller pieces, and how to manage application state using the ReaderT pattern that is ubiquitous in Haskell web development.

The book then moves into production concerns. We integrate PostgreSQL for persistent storage with proper connection pooling and transaction management. We implement authentication using JWT tokens and authorization based on user roles. We add logging, structured error handling, CORS support, and other middleware essential for real-world applications. You will learn how to generate OpenAPI documentation that developers can explore interactively, and how to write comprehensive tests at every level from unit tests of individual handlers to integration tests against a running server.

Finally, we cover deployment: containerizing your application with Docker, configuring reverse proxies like Nginx for HTTPS termination, setting up CI/CD pipelines, and operational best practices for keeping your API healthy in production. Throughout the book, we build toward one substantial reference application that ties all these pieces together into a coherent whole.

This is not a book about Haskell theory or type-level programming abstractions for their own sake. Every concept is introduced because it solves a concrete problem in web development. When we discuss monad transformers, it is because you need them to thread configuration and database connections through your handlers cleanly. When we explore DataKinds and TypeOperators, it is because they enable Servant's type-level DSL. The theory serves the practice.

By the time you finish this book, you will not only know how to use Servant. You will understand why it works the way it does, when it is the right tool for your project, and how to adapt its patterns to problems we may not have covered explicitly. Let us begin.

Chapter 1: The Haskell Web Ecosystem and Servant's Philosophy

The Landscape of Haskell Web Frameworks

Haskell has a rich ecosystem of web frameworks, each with different design philosophies and trade-offs. Understanding this landscape helps you appreciate what Servant brings to the table and when it is the right choice for your project.

The most prominent alternatives to Servant are Yesod, Scotty, Snap, and Spock. Each represents a different approach to solving the problems of web development in Haskell.

Yesod is the most full-featured Haskell web framework. It provides an opinionated architecture with built-in support for forms, templates, authentication, persistent storage via the Persistent library, and internationalization. Yesod uses Template Haskell extensively to provide type-safe URLs and compile-time checks on form handling. It is designed as a batteries-included framework where many decisions are made for you. This makes Yesod excellent for large applications with traditional server-rendered views but can feel heavyweight if you only need a JSON API.

Scotty takes the opposite approach. Inspired by Sinatra in Ruby, it provides minimal routing primitives and leaves most architectural decisions to you. You define routes as functions that take request information and produce responses. Scotty is simple to learn and great for small services or prototypes, but it lacks type safety at the routing level. A typo in a route pattern is not caught until runtime.

Snap and Spock occupy middle ground. Snap is one of Haskell's older web frameworks with a comprehensive ecosystem but a steeper learning curve due to its monadic design and extensive use of lenses. Spock provides type-safe routing through string-based route definitions while remaining lightweight and flexible.

Servant distinguishes itself from all of these by focusing on a single idea: the API description as a type-level specification. Rather than providing opinions about forms, templates, or database access, Servant concentrates on making the contract between your server and its clients precise, verifiable, and reusable. This focus makes it particularly well-suited for building RESTful APIs and microservices where clear contracts matter more than server-rendered HTML views.

What Makes Servant Different

The defining characteristic of Servant is that your API is not described with strings, macros, or runtime configuration. It is described as a Haskell type. Consider this simple example:

```
type API = "hello" :> Get '[PlainText] Text
```

This single line declares an endpoint at the path /hello that responds to GET requests with plain text. But this is not merely documentation written in Haskell syntax. This type is interpreted by Servant's type system to:

1. Generate a routing table that matches incoming HTTP requests to handlers
2. Define the exact signature your handler function must have
3. Derive client functions that call this endpoint correctly
4. Produce documentation describing this endpoint

If you write a handler with the wrong return type, or if you try to add a query parameter that is not declared in the API type, the compiler catches the error before your code runs. This is fundamentally different from frameworks where routes are strings matched at runtime.

This type-level approach also enables something unique: sharing the same API description between server and client code in the same project or across projects. When you change the API on the server side by modifying its type, every place that depends on that API must be updated to match. The compiler enforces consistency automatically. There is no risk of documentation rot where your OpenAPI spec describes endpoints that no longer exist.

Servant was originally developed at Zalora, an e-commerce company, and later became part of the MIT Lincoln Laboratory's portfolio before moving to community maintenance under the haskell-servant organization on GitHub. Its design emerged from practical experience building large-scale web services where API contracts needed to be reliable and maintainable across multiple teams and languages.

Type-Level API Descriptions as Single Source of Truth

The phrase "single source of truth" appears frequently in software engineering discussions, but Servant implements it literally through the type system. The API type serves simultaneously as:

- A specification that stakeholders can read to understand what endpoints exist
- A compiler-checked contract that server handlers must satisfy
- Input for client code generation tools
- Input for documentation generation tools
- Input for testing frameworks that validate API behavior

This multi-purpose use of a single declaration eliminates entire categories of bugs. Consider what happens when you rename a field in your response JSON. In most frameworks, you update the handler and hope to remember to update the documentation and any client code. With Servant, if the server and client share the same API type, renaming the field breaks compilation on both sides until everything is consistent.

The type-level DSL uses combinators that correspond directly to HTTP concepts. Path segments are combined with an infix operator. HTTP methods like GET and POST are represented as types. Query parameters, path captures, request bodies, and response headers each have their own combinator. These building blocks compose naturally, allowing you to describe complex APIs while keeping the type-level code readable.

For example, a more realistic API might look like this:

```
type UserAPI = "users" :> Get '[JSON] [User]
               :<|> "users" :> Capture "id" UserId :> Get '[JSON] User
               :<|> "users" :> ReqBody '[JSON] CreateUserRequest :> Post '[JSON]
               ↪  User
```

This declares three endpoints: one for listing all users, one for getting a specific user by ID from the path, and one for creating a new user with a JSON body. Each endpoint's structure is visible in its type, and Servant ensures handlers match these declarations exactly.

How Servant Splits Its Responsibilities Across Packages

Servant is not a single monolithic package. It is organized as a family of related packages that separate concerns cleanly. Understanding this architecture helps you choose which packages your project needs and why certain functionality lives where it does.

The core `servant` package defines the type-level DSL itself. It contains the combinators like Get, Post, Capture, QueryParam, and the operators :> and :<|>. Crucially, this package has no dependencies on specific web servers or HTTP clients. It is purely about describing APIs at the type level. This design allows the API description to be shared across server code, client code, and documentation generators without pulling in unnecessary dependencies.

The `servant-server` package implements the server side of the story. It defines the Handler monad where your endpoint logic runs, the serve function that turns an API type and handlers into a WAI Application, and all the routing machinery that matches incoming requests to the correct handler. It depends on Warp as its default web server but can work with other WAI-compatible servers.

The `servant-client` package generates client functions from API types. Given the same API type used on the server, it produces Haskell functions that construct HTTP requests with the correct paths, methods, headers, and bodies. These client functions run in a ClientM monad that handles HTTP communication using the http-client library.

Additional packages extend Servant's capabilities:

- `servant-auth` and `servant-auth-server` provide authentication combinators for JWT tokens, cookies, and basic authentication

- `servant-swagger` generates OpenAPI 2.0 (Swagger) specifications from API types
- `servant-openapi3` generates OpenAPI 3.0 specifications
- `servant-docs` produces human-readable documentation in various formats

This modular architecture is intentional. Your API description package might depend only on the core servant library, making it lightweight and easy to share. Server code adds a dependency on servant-server. Client code adds servant-client. Documentation tools add their respective packages. Each piece can evolve independently while sharing the same type-level foundation.

When to Choose Servant and When Not To

Servant excels in specific scenarios but is not universally optimal. Understanding its strengths and limitations helps you make better technology choices.

Choose Servant when:

- You are building a RESTful API or microservice where the contract between server and client is central to your architecture
- You want compile-time guarantees that your handlers match your API specification
- You plan to generate clients in Haskell from the same API description
- You value type safety over rapid prototyping speed
- Your team is comfortable with (or willing to learn) Haskell's type system
- You need documentation that cannot drift out of sync with implementation

Servant may not be the best choice when:

- You are building a traditional server-rendered web application with HTML views and forms. Yesod or another full-stack framework would be more appropriate.
- You need rapid prototyping with minimal Haskell knowledge. Scotty has a gentler learning curve for simple routing needs.
- Your primary concern is GraphQL rather than REST. While Servant can handle GraphQL endpoints, dedicated libraries may serve you better.

- You are working in a team unfamiliar with functional programming and type-level programming. The cognitive overhead of Servant's approach may slow initial development.

Even when Servant is the right choice, it does not solve every problem in web development. It focuses on API description and routing. You still need to choose libraries for database access, authentication implementation, logging, configuration management, and deployment. The ecosystem provides excellent options for all of these, but they are separate concerns that you must integrate yourself.

Servant is a framework for APIs first and foremost. If your project's core challenge is defining and maintaining reliable API contracts, Servant's type-level approach provides benefits that are difficult to match with other tools. If your challenges lie elsewhere, you may find more value in a different part of the Haskell ecosystem.

Chapter 2: Getting Started with Haskell and Your Development Environment

Installing GHC and Choosing a Version

Before writing any Haskell code, you need the Glasgow Haskell Compiler (GHC) and supporting tooling installed on your system. The recommended approach is to use GHCup, a cross-platform installer that manages all Haskell tools consistently.

Install GHCup by following the instructions at https://www.haskell.org/ghcup/. On most systems this is a single command:

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

After installation, install GHC version 9.8 or later along with Cabal and HLS (Haskell Language Server):

```
ghcup install ghc 9.8.4
ghcup set ghc 9.8.4
ghcup install cabal 3.12
ghcup install hls 2.10
```

GHC 9.8 is a solid choice for production work as of this writing. It is stable, well-tested, and supported by the entire Servant ecosystem. The Servant packages are tested with GHC versions from 9.2 through 9.12+, so you have flexibility if your project has specific version requirements. Newer GHC versions bring performance improvements and language features, but they also occasionally introduce breaking changes in type inference or error messages that can affect existing codebases.

If you prefer the absolute latest stable compiler, GHC 9.10 or 9.12 are available through GHCup. The examples in this book target GHC 9.8 but should work with any version from 9.6 upward with minimal adjustments.

Build Tools: Cabal versus Stack in Modern Practice

Haskell projects use build tools to manage dependencies, compile code, and run executables. Two tools dominate the ecosystem: Cabal and Stack. Understanding their relationship and current status helps you make informed choices.

Cabal originally referred only to a library for describing Haskell packages but now encompasses the entire package management infrastructure. The .cabal file format defines package metadata including name, version, dependencies, and build instructions. The cabal-install tool (invoked as `cabal`) reads these files to resolve dependencies and build projects. Modern Cabal (version 3.x) has matured significantly and addresses many historical pain points around dependency resolution and reproducibility.

Stack emerged around 2015 to solve problems that existed with older Cabal versions, particularly "dependency hell" where different projects required incompatible package versions. Stack introduced Stackage snapshots: curated sets of packages known to work together at specific versions. This provided reproducible builds without requiring developers to manually resolve conflicts. Stack also managed GHC installation automatically, which was convenient for newcomers.

The landscape has shifted since Stack's peak popularity. Modern Cabal now handles dependency resolution robustly through its solver and supports freeze files (cabal.project.freeze) that pin exact package versions for reproducibility, similar to Stackage snapshots. Meanwhile, Stack's development has slowed relative to Cabal's active maintenance. GHCup now manages GHC installation cleanly, removing one of Stack's key advantages.

For new projects in 2026, the recommendation is:

- Use modern Cabal (3.10+) as your primary build tool for most projects
- Use Stack if you are joining an existing project that uses it or if you specifically want Stackage snapshot guarantees without manual configuration
- Both tools use the same .cabal file format, so migrating between them is straightforward

This book uses Cabal for all examples because it is the official Haskell build tool with active development and broader ecosystem integration. The concepts transfer directly to Stack users.

Your First Haskell Project Structure

Let us create a minimal Servant project from scratch to establish patterns you will use throughout this book. Start by creating a directory and initializing a Cabal project:

```
mkdir servant-todo-api
cd servant-todo-api
cabal init --lib --exe
```

The `--lib` flag creates a library component for shared code, and `--exe` creates an executable target. Cabal will ask you a series of questions about your project. Accept the defaults or customize as needed. The resulting structure looks like this:

```
servant-todo-api/
├── servant-todo-api.cabal
├── cabal.project
├── src/
│   └── ServantTodoAPI.hs
└── app/
    └── Main.hs
```

The .cabal file describes your package. Open it and add the dependencies you need for a Servant application:

```
cabal-version:      >= 2.4
name:               servant-todo-api
version:            0.1.0.0
synopsis:           A Todo API built with Servant
license:            MIT
author:             Your Name
maintainer:         your.email@example.com

common deps
    build-depends:      base >= 4.14 && < 5
                      , servant >= 0.20
                      , servant-server >= 0.20
                      , warp >= 3.4
                      , aeson >= 2.0
                      , text
                      , bytestring

```

```
library
    other-modules:      ServantTodoAPI.API
                      , ServantTodoAPI.Types
                      , ServantTodoAPI.Server
    ghc-options:        -Wall -Wcompat -Wincomplete-record-updates
    build-depends:      deps

executable servant-todo-api-exe
    main-is:            Main.hs
    other-modules:      Paths_servant_todo_api
    ghc-options:        -threaded -rtsopts -with-rtsopts=-N
    build-depends:      deps
                      , servant-todo-api
```

Several things deserve explanation here. The `common deps` stanza defines dependencies shared between the library and executable, avoiding duplication. The GHC options enable warnings that help catch bugs early: `-Wall` enables most warnings, `-Wcompat` warns about code that may break with future GHC versions, and `-Wincomplete-record-updates` catches incomplete record field updates.

The `build-depends` field specifies version constraints using the Package Versioning Policy (PVP). The pattern `>= 0.20 && < 0.21` means "any version starting with 0.20 but before 0.21." PVP conventions use the first two digits of a version number to indicate major compatibility boundaries. A change in either of these digits may introduce breaking changes.

The executable links against both its dependencies and the library component (`servant-todo-api`), allowing it to import your application's modules.

Now create the module structure. Cabal initialized ServantTodoAPI.hs as a placeholder. Delete it and create proper modules:

```
mkdir -p src/ServantTodoAPI
touch src/ServantTodoAPI/API.hs
touch src/ServantTodoAPI/Types.hs
touch src/ServantTodoAPI/Server.hs
```

This organization separates concerns: API.hs defines the type-level API description, Types.hs contains data types used by the API, and Server.hs implements handlers. This separation scales well as your application grows.

Essential Development Tooling and Extensions

Beyond GHC and Cabal, several tools significantly improve the Haskell development experience. The Haskell Language Server (HLS) provides IDE-like features including type information on hover, go-to-definition, autocomplete, and inline error display. Install it via GHCup as shown earlier, then configure your editor.

For Visual Studio Code, install the "Haskell" extension by haskell-language-server. It automatically connects to HLS and provides comprehensive support. For Neovim, use plugins like nvim-haskell or lspconfig with Haskell configured. Emacs users can use haskell-mode with lsp-mode or the dedicated lhaskell-mode.

For interactive development and experimentation, GHCi (the Haskell REPL) is invaluable. Run it within your project context using:

```
cabal repl servant-todo-api
```

This loads your library's modules into GHCi with all dependencies available, allowing you to test functions interactively without restarting the compiler each time.

For fast feedback during development, ghcid provides automatic recompilation as you edit files. Install it via Cabal:

```
cabal install ghcid
```

Then run it in your project directory:

```
ghcid -c "cabal repl servant-todo-api"
```

Now every time you save a file, GHCi reloads with your changes and runs any commands you have queued. This is particularly useful for developing Servant applications where type errors can be verbose and understanding them iteratively helps learning.

Your editor should also be configured to use ormolu or fourmolu for formatting Haskell code consistently. These formatters eliminate debates about style

by applying a single canonical format. Fourmolu extends ormolu with support for language extensions commonly used in Servant applications. Install it via GHCup:

```
ghcup install fourmolu
```

Compiling, Running, and Debugging Basics

With your project structure in place, let us verify everything works by building a minimal application. First, update app/Main.hs with a simple starting point:

```
module Main where

import qualified Network.Wai.Handler.Warp as Warp

main :: IO ()
main = do
    putStrLn "Starting server on port 3000..."
    Warp.run 3000 $ \_req respond ->
        respond $ responseLBS status200 [("Content-Type", "text/plain")] "Hello
        ↪  from Servant!"
```

Add the required imports at the top:

```
import Network.HTTP.Types (status200)
import qualified Data.ByteString.Lazy.Char8 as LBS
import Network.Wai (responseLBS)
```

Build and run with:

```
cabal build all
cabal run servant-todo-api-exe
```

Visit http://localhost:3000 in your browser to see the response. This is a bare WAI application without Servant yet, but it confirms your environment works correctly.

To debug Haskell programs effectively, understand these techniques:

- Use `putStrLn` for quick debugging output during development. Haskell's lazy evaluation means print statements may appear at unexpected times, so use them cautiously.
- For more reliable logging, use a proper logging library like fast-logger or the logging package, which we will cover later in this book.
- GHCi allows you to load modules and evaluate expressions interactively. Use `:t` to check types of expressions, which is invaluable when working with Servant's complex type-level code.
- When encountering type errors, read them carefully from top to bottom. GHC often provides excellent suggestions for fixing issues, especially with recent versions. The first error message may cascade into many others, so fix the first one and recompile before worrying about subsequent errors.

For runtime debugging, Haskell programs can be compiled with profiling support using `cabal build --enable-profiling`. This generates additional information that tools like GHC's built-in profiler can use to analyze memory allocation and CPU usage. We will cover profiling in more detail when discussing production performance optimization.

Chapter 3: The Haskell Foundations You Need for Servant

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Types, Type Inference, and Algebraic Data Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Functions, Partial Application, and Currying

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Type Classes: Polymorphism with Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

The Maybe and Either Types for Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Monads: Chaining Computations with Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Monad Transformers: Layering Effects

Chapter 4: Your First Servant API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Defining Your First Type-Level API Route

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

The Handler Monad and Returning Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Serving Your API with Warp

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Testing Endpoints with curl and HTTP Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Understanding How Servant Routes Requests to Handlers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Chapter 5: Deep Dive into Servant Combinators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

HTTP Methods: Get, Post, Put, Patch, Delete

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Path Parameters with Capture and CaptureAll

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Query Parameters: Single, Multiple, and Flags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Request Bodies with ReqBody

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Response Types, Status Codes, and Content Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Combining Routes with the Left Alternation Operator

Chapter 6: JSON Serialization with Aeson

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

The Aeson Ecosystem: ToJSON, FromJSON, and Value

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Manual Instance Derivation for Full Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Automatic Deriving with Generic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Custom Serialization Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Handling Partial Updates and Optional Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Common JSON Pitfalls in Servant Applications

Chapter 7: API Composition and Modular Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Composing APIs with the Alternation Operator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Nested Routes and Sub-APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Reusable API Type Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Versioning Strategies for Evolving APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Custom Servant Combinators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Module Organization for Large Projects

Chapter 8: Application Environments and Dependency Injection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

The Problem of Global State in Web Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

ReaderT as an Environment Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Designing Your Application Environment Record

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

hoistServer: Adapting Handler Monads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Configuration Management Across Environments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Dependency Injection Patterns in Servant

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Chapter 9: Error Handling, Validation, and Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Servant's Structured Error System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Custom Error Types and Status Codes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Input Validation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Logging with Middleware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Request IDs and Traceability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Exception Handling in Handlers

Chapter 10: Persistence with PostgreSQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Choosing a Database Library: postgresql-simple versus Others

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Connection Pooling with resource-pool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Transactions and Isolation Levels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Database Migrations with migrate

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Repository Layer Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Mapping Between Database and Domain Models

Chapter 11: Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Security Fundamentals for Web APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Password Hashing with Argon2 or bcrypt

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

JWT-Based Token Authentication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Session and Cookie-Based Authentication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Authorization: Roles, Permissions, and Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Protecting Specific Endpoints with Servant Combinators

Chapter 12: Building the Reference Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Application Architecture Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Directory Structure and Module Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Domain Models and Database Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

The Complete API Definition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Handler Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Configuration and Application Bootstrap

Chapter 13: Clients, Documentation, and External Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Generated Servant Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Handwritten Client Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Calling External APIs from Your Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

OpenAPI and Swagger Documentation Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Serving Interactive API Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Chapter 14: Testing Your Servant Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Unit Testing Handlers in Isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Property-Based Testing with QuickCheck

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Integration Tests Against a Running Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Test Databases and Schema Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Mocking External Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Organizing Your Test Suite

Chapter 15: Middleware, CORS, and Cross-Cutting Concerns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

WAI Middleware Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Handling CORS Properly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Compression and Performance Middleware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Timeouts and Rate Limiting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Custom Middleware for Logging and Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Ordering and Composing Middleware

Chapter 16: Observability, Performance, and Production Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Metrics Collection and Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Distributed Tracing Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Profiling Haskell Web Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Caching Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Concurrency and Resource Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Graceful Shutdown and Reloading

Chapter 17: Deployment and DevOps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Containerizing with Docker

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Multi-Stage Builds for Small Images

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Reverse Proxies: Nginx and Traefik

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

HTTPS Termination

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

CI/CD Pipelines with GitHub Actions or GitLab CI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Configuration Management in Production

Conclusion: The Servant Way Forward

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Servant Core Documentation and Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

JSON Serialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Web Server and WAI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Database Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

OpenAPI and Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Build Tools and Tooling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

GHC and Haskell Language

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

External Articles and Tutorials

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Academic Papers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.

Version Information Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingwebapplicationswithservantinhaskell.