

Created by Randy Fadler

August 2025

**Executive Summary: Introduction to F# Booklet**

The "Introduction to F# Booklet" serves as a comprehensive guide designed to equip developers, students, and IT professionals with the knowledge and skills to master F#, a robust, functional-first programming language within the .NET ecosystem. Released on August 30, 2025, this booklet spans 11 chapters, offering a progressive learning journey from foundational concepts to building a full-stack web application. It leverages F#'s strengths—type safety, immutability, and concurrency—to address modern software development challenges.

The booklet begins with an overview of F#’s history, syntax, and functional programming principles, establishing a solid base for learners. Subsequent chapters introduce practical applications, including pattern matching, object-oriented integration, and asynchronous programming, with hands-on examples like a console-based task manager. Advanced topics such as domain-driven design (DDD), Railway-Oriented Programming (ROP) for error handling, and parallel processing are explored, enhancing code robustness and scalability. The culmination in Chapter 11 delivers a full-stack web application using Giraffe for the backend and React with Tailwind CSS for the frontend, integrating the task manager with a modern, responsive user interface.

Key highlights include:

* **Educational Depth**: Detailed explanations, analogies, and comprehensive examples ensure accessibility for beginners while challenging experienced developers.
* **Project-Based Learning**: A task management application evolves across chapters, reinforcing concepts with real-world applicability.
* **Modern Relevance**: Incorporates cutting-edge techniques like asynchronous workflows, parallel sequences, and actor-based concurrency, tailored to multi-core and cloud environments.
* **Practical Tools**: Includes setup instructions, FSI experiments, and unit tests, with a futuristic booklet cover design reflecting F#'s innovative spirit.

Targeted at individuals seeking to leverage F# for high-performance, maintainable software, this booklet is ideal for self-study or corporate training. It prepares readers to build scalable applications, from console tools to web platforms, and positions F# as a strategic asset in the 2025 tech landscape. The accompanying web app, accessible via local deployment, provides a tangible deliverable, while the open-source code fosters community engagement and further development.

Table of Contents

[F# Programming Language Booklet Outline 6](#_Toc207450157)

[1. Introduction to F# 6](#_Toc207450158)

[2. Getting Started with F# 6](#_Toc207450159)

[3. Core Concepts of F# 6](#_Toc207450160)

[4. F# Syntax and Basics 7](#_Toc207450161)

[5. Functional Programming in F# 7](#_Toc207450162)

[6. Object-Oriented Programming in F# 7](#_Toc207450163)

[7. Advanced F# Features 7](#_Toc207450164)

[8. Practical F# Applications 8](#_Toc207450165)

[9. F# Ecosystem and Community 8](#_Toc207450166)

[10. Best Practices and Tips 8](#_Toc207450167)

[11. Comprehensive F# Project 8](#_Toc207450168)

[12. Conclusion 9](#_Toc207450169)

[13. Appendices 9](#_Toc207450170)

[1. Introduction to F# 10](#_Toc207450171)

[Overview of F# as a Functional-First Programming Language 10](#_Toc207450172)

[History and Development by Microsoft Research 10](#_Toc207450173)

[Benefits of Using F# 11](#_Toc207450174)

[Ongoing Project Overview 12](#_Toc207450175)

[Overview of the Comprehensive Project 12](#_Toc207450176)

[What’s Next? 13](#_Toc207450177)

[2. Getting Started with F# 14](#_Toc207450178)

[Installation and Setup 14](#_Toc207450179)

[Step-by-Step Installation 14](#_Toc207450180)

[Common Pitfalls: 15](#_Toc207450181)

[Writing Your First F# Program (Hello World Example) 16](#_Toc207450182)

[Basic Project Structure and Tools 20](#_Toc207450183)

[3. Core Concepts of F# 23](#_Toc207450184)

[Functional Programming Principles in F# 23](#_Toc207450185)

[Key Principles of FP in F# 23](#_Toc207450186)

[Why FP in F#? 24](#_Toc207450187)

[Static Typing Benefits 25](#_Toc207450188)

[Pattern Matching and Discriminated Unions 26](#_Toc207450189)

[Functions as First-Class Citizens 27](#_Toc207450190)

[Why First-Class? 27](#_Toc207450191)

[Ongoing Project: Defining Basic Task Types and Simple Functions 27](#_Toc207450192)

[4. F# Syntax and Basics 30](#_Toc207450193)

[Variables and Bindings 30](#_Toc207450194)

[Mutable Bindings 30](#_Toc207450195)

[Shadowing 31](#_Toc207450196)

[Type Inference 32](#_Toc207450197)

[Control Flow 32](#_Toc207450198)

[Match Expressions 33](#_Toc207450199)

[Lists, Sequences, and Arrays 34](#_Toc207450200)

[Records 36](#_Toc207450201)

[5. Functional Programming in F# 41](#_Toc207450202)

[Higher-Order Functions and Function Composition 41](#_Toc207450203)

[Function Composition 42](#_Toc207450204)

[Pipelining and the Forward Operator (|>) 42](#_Toc207450205)

[Working with Immutable Data Structures 44](#_Toc207450206)

[Ongoing Project: Adding Functional Operations for Task Filtering and Manipulation 46](#_Toc207450207)

[6. Object-Oriented Programming in F# 52](#_Toc207450208)

[Classes, Interfaces, and Inheritance 52](#_Toc207450209)

[Interfaces 53](#_Toc207450210)

[Inheritance 53](#_Toc207450211)

[Records vs. Classes 54](#_Toc207450212)

[Encapsulation and Access Modifiers 56](#_Toc207450213)

[7. Advanced F# Features 64](#_Toc207450214)

[Type Providers 64](#_Toc207450215)

[Computation Expressions 65](#_Toc207450216)

[Units of Measure 66](#_Toc207450217)

[Active Patterns 67](#_Toc207450218)

[Partial Active Pattern 68](#_Toc207450219)

[Quotations and Meta-Programming 68](#_Toc207450220)

[Ongoing Project: Enhancing the Task Manager with Async and Data Querying 69](#_Toc207450221)

[8. Practical F# Applications 79](#_Toc207450222)

[Data Processing with F# 79](#_Toc207450223)

[Unit Testing in F# 81](#_Toc207450224)

[Why Giraffe? 84](#_Toc207450225)

[Running the Projects 94](#_Toc207450226)

[9. Concurrent and Parallel Programming in F# 95](#_Toc207450227)

[Asynchronous Workflows 95](#_Toc207450228)

[Task Parallel Library (TPL) 97](#_Toc207450229)

[10. Domain-Driven Design and Error Handling in F# 115](#_Toc207450230)

[Domain-Driven Design in F# 115](#_Toc207450231)

[F# and DDD 115](#_Toc207450232)

[Error Handling with Railway-Oriented Programming (ROP) 116](#_Toc207450233)

[Advanced Pattern Matching 118](#_Toc207450234)

[11. Building a Full-Stack Web Application in F# 138](#_Toc207450235)

[Overview of the Full-Stack Application 138](#_Toc207450236)

[Web API: Giraffe Backend 146](#_Toc207450237)

[Experiment in FSI 164](#_Toc207450238)

# F# Programming Language Booklet Outline

This outline incorporates an ongoing project that builds incrementally throughout the core learning chapters (Chapters 2-7). The ongoing project will be a simple console-based task management application, where each chapter adds new features or refactors existing code to demonstrate the concepts introduced. This hands-on approach reinforces learning by applying concepts immediately.

A separate, more comprehensive project—a full-stack web application using F# for backend and data processing—will be presented as the final main chapter before the conclusion.

## 1. Introduction to F#

* Overview of F# as a functional-first programming language
* History and development by Microsoft Research
* Key features: functional, object-oriented, and imperative programming
* Benefits of using F# (conciseness, type safety, performance)
* Introduction to the ongoing project: Building a console-based task manager
* Overview of the comprehensive project to be built at the end

## 2. Getting Started with F#

* Installation and setup (F# compiler, .NET SDK, IDEs like Visual Studio Code)
* Writing your first F# program (Hello World example)
* Overview of F# Interactive (FSI) for rapid prototyping
* Basic project structure and tools (e.g., .fs, .fsproj files)
* Ongoing Project: Setting up the initial project structure and basic entry point

## 3. Core Concepts of F#

* Functional programming principles in F#
* Immutability and expressions
* Type inference and static typing
* Pattern matching and discriminated unions
* Functions as first-class citizens
* Ongoing Project: Defining basic task types and simple functions

## 4. F# Syntax and Basics

* Variables and bindings (let, mutable)
* Basic data types (int, string, list, array, etc.)
* Control flow (if-then-else, match expressions)
* Lists, sequences, and arrays: working with collections
* Tuples and records
* Ongoing Project: Implementing task storage using lists and records

## 5. Functional Programming in F#

* Higher-order functions and function composition
* Pipelining and the forward operator (|> )
* Recursion and tail-call optimization
* Working with immutable data structures
* Map, filter, and fold operations
* Ongoing Project: Adding functional operations for task filtering and manipulation

## 6. Object-Oriented Programming in F#

* Classes, interfaces, and inheritance
* Records vs. classes
* Interoperability with C# and .NET libraries
* Encapsulation and access modifiers
* Ongoing Project: Extending the task manager with OOP elements, such as classes for user interactions

## 7. Advanced F# Features

* Type providers for data access
* Computation expressions (e.g., async, query)
* Units of measure for type-safe calculations
* Active patterns
* Quotations and meta-programming
* Ongoing Project: Enhancing the app with async operations and data querying

## 8. Practical F# Applications

* Building web applications with F# (e.g., using Giraffe or Suave)
* Data science and machine learning with F#
* Domain-driven design with F#
* Scripting and automation tasks
* Case studies: real-world F# projects

## 9. F# Ecosystem and Community

* Popular F# libraries and frameworks (FSharp.Data, FsUnit, etc.)
* Integration with .NET ecosystem
* Community resources: F# Software Foundation, forums, and blogs
* Contributing to open-source F# projects

## 10. Best Practices and Tips

* Writing idiomatic F# code
* Debugging and testing F# applications
* Performance optimization techniques
* Common pitfalls and how to avoid them
* Applying best practices to the ongoing project

## 11. Comprehensive F# Project

* Overview of the project: Building a full-stack task management web app
* Step-by-step implementation using concepts from previous chapters
* Integrating functional and OOP paradigms
* Using advanced features like async and type providers
* Deployment and final touches
* Extensions and variations for further exploration

## 12. Conclusion

* Recap of F# strengths and use cases
* Review of the ongoing and comprehensive projects
* Encouragement to explore F# further
* Resources for continued learning (books, tutorials, online courses)

## 13. Appendices

* Glossary of F# terms
* Cheat sheet for F# syntax
* References and further reading
* Sample code snippets from the ongoing and comprehensive projects

# 1. Introduction to F#

F# is a powerful, functional-first programming language designed to be concise, expressive, and type-safe. Developed by Microsoft Research, it combines the elegance of functional programming with the robustness of the .NET ecosystem, making it suitable for a wide range of applications, from web development to data science and scripting. This chapter introduces F#, its history, key features, and the exciting projects you'll build in this booklet to master its concepts.

## Overview of F# as a Functional-First Programming Language

F# is a multi-paradigm language that emphasizes functional programming but also supports object-oriented and imperative styles. It runs on the .NET runtime, enabling seamless integration with other .NET languages like C#. Its functional-first approach encourages writing code that is declarative, immutable, and focused on expressions rather than statements, leading to fewer bugs and easier reasoning about code behavior.

Key characteristics of F# include:

* **Conciseness**: F# reduces boilerplate code, allowing developers to express complex logic succinctly.
* **Type Safety**: Its strong static type system catches errors at compile time without requiring excessive type annotations, thanks to type inference.
* **Interoperability**: F# works seamlessly with .NET libraries and C#, making it ideal for enterprise and cross-platform development.
* **Versatility**: It supports functional constructs like pattern matching and higher-order functions, alongside object-oriented features like classes and interfaces.

F# is particularly well-suited for domains requiring robust data processing, such as financial systems, scientific computing, and machine learning, as well as rapid prototyping and scripting.

## History and Development by Microsoft Research

F# was created by Don Syme at Microsoft Research in the early 2000s, with its first release in 2005. Inspired by functional languages like ML, OCaml, and Haskell, F# was designed to bring functional programming to the .NET platform. Over the years, it has evolved into a mature language, with significant updates aligning it with modern .NET advancements, such as .NET Core and .NET 8. The F# community, supported by the F# Software Foundation, has played a crucial role in its growth, fostering open-source contributions and widespread adoption.

**Key Features**

F# offers a rich set of features that make it unique:

* **Functional Programming**: Supports immutable data, first-class functions, and pattern matching for concise and robust code.
* **Object-Oriented Programming**: Provides classes, interfaces, and inheritance for compatibility with .NET and structured design.
* **Imperative Programming**: Allows mutable state and loops when needed, offering flexibility for practical tasks.
* **Type Inference**: Reduces the need for explicit type annotations while maintaining strong typing.
* **Computation Expressions**: Simplifies asynchronous programming, data querying, and other workflows.
* **Type Providers**: Enables seamless integration with external data sources, such as databases or APIs, with compile-time type safety.

These features make F# a versatile tool for both small scripts and large-scale applications.

## Benefits of Using F#

F# offers several advantages for developers:

* **Conciseness**: Its syntax minimizes boilerplate, enabling faster development and easier maintenance.
* **Type Safety**: The type system prevents common errors, such as null reference exceptions, through features like option types.
* **Performance**: Running on .NET, F# delivers high performance, comparable to C#, with optimizations for functional code.
* **Productivity**: Tools like F# Interactive (FSI) allow rapid experimentation and prototyping.
* **Interoperability**: F# integrates effortlessly with existing .NET codebases, libraries, and tools, making it practical for enterprise use.

These benefits make F# an excellent choice for developers seeking a balance of elegance and practicality.

**Introduction to the Ongoing Project: Building a Console-Based Task Manager**

To make learning F# hands-on, this booklet includes an ongoing project that evolves through the core chapters (2–7). You’ll build a console-based task management application, starting with a simple task list and gradually adding features like filtering, prioritization, and asynchronous operations. Each chapter will introduce new F# concepts and apply them to the project, reinforcing your understanding through practical application. By the end, you’ll have a fully functional console app showcasing F#’s core features.

## Ongoing Project Overview

The task manager will:

* Store tasks with attributes like description, priority, and due date.
* Support adding, removing, and updating tasks.
* Allow filtering and sorting tasks based on criteria like priority or status.
* Use functional programming for data manipulation and object-oriented features for user interaction.
* Incorporate advanced features like async operations for file I/O or external data access.

In Chapter 2, you’ll set up the project and create the initial task structure, building on it as you learn new concepts.

## Overview of the Comprehensive Project

As a capstone, Chapter 11 will guide you through building a full-stack task management web application using F# and a web framework like Giraffe. This project will combine everything you’ve learned, including:

* A backend written in F# to handle task data and business logic.
* Functional programming for data processing and transformations.
* Object-oriented design for structuring the application.
* Advanced features like async workflows and type providers for data integration.
* Deployment to a web server for real-world use.

This project will demonstrate F#’s power in building modern, scalable applications and prepare you for real-world development scenarios.

## What’s Next?

In Chapter 2, you’ll set up your F# development environment, write your first F# program, and begin the ongoing task manager project. Get ready to dive into F#’s syntax and start coding!

# 2. Getting Started with F#

Welcome to the first hands-on chapter of this F# booklet! Here, we'll guide you through setting up your development environment, writing your inaugural F# program, exploring the interactive features of F#, and understanding the basic structure of an F# project. As a functional-first language on the .NET platform, getting started with F# is straightforward, especially if you're familiar with other .NET languages. We'll explain each step in depth, like a textbook, with detailed reasoning, examples, and breakdowns to ensure you grasp not just the "how" but also the "why."

By the end of this chapter, you'll have a working F# environment and the foundation for our ongoing project: a console-based task manager. Let's dive in!

## Installation and Setup

To develop in F#, you need the .NET Software Development Kit (SDK), which includes the F# compiler and runtime. F# is fully integrated into .NET, so installing the .NET SDK gives you everything required to compile and run F# code. As of August 2025, the latest stable version is .NET 9, but always check for updates.

## Step-by-Step Installation

1. **Download the .NET SDK**:
   * Visit the official Microsoft .NET download page: <https://dotnet.microsoft.com/download>.
   * Select the SDK for your operating system (Windows, macOS, or Linux). For most users, the current version (.NET 9) is recommended for new projects.
   * Why .NET SDK? The SDK includes the runtime (for running apps), compilers (including F#), and tools like dotnet CLI for project management. Installing just the runtime won't allow you to build code.
2. **Install the SDK**:
   * Run the installer and follow the prompts. On Windows, it's an .exe file; on macOS, a .pkg; on Linux, use package managers like apt or yum as per the instructions.
   * After installation, open a terminal or command prompt and verify by running:

dotnet --version

This should output something like `9.0.304` (the exact version may vary). If it does, the installation succeeded.

1. **Choose an Integrated Development Environment (IDE)**:
   * **Visual Studio Code (VS Code)**: Free, lightweight, and highly recommended for F# development.
     + Download from <https://code.visualstudio.com/>.
     + Install the Ionide extension: Open VS Code, go to Extensions (Ctrl+Shift+X), search for "Ionide-fsharp", and install it.
     + Why Ionide? It provides F#-specific features like IntelliSense (auto-completion), debugging, project exploration, and F# Interactive integration. It's maintained by the F# community and enhances productivity with real-time feedback.
   * **Visual Studio**: If you prefer a full IDE, download Visual Studio Community (free) from <https://visualstudio.microsoft.com/>. During installation, select the ".NET desktop development" workload, which includes F# support.
     + Visual Studio offers advanced debugging and project templates but is heavier than VS Code.
   * **Other Options**: Rider by JetBrains or even text editors like Vim with plugins, but for beginners, VS Code with Ionide is ideal due to its simplicity and power.
2. **Verify F# Support**:
   * In your terminal, run:

dotnet new console -lang "F#" -o MyFirstFSharpApp

This creates a new F# console project. If it succeeds without errors, your setup is complete.

* Explanation: The dotnet new command uses templates to scaffold projects. The -lang "F#" flag specifies F# (default is C#). This tests both the SDK and F# compiler.

## Common Pitfalls:

* Ensure your PATH environment variable includes the .NET installation directory (usually handled by the installer).
* On Linux/macOS, you might need to restart your terminal for changes to take effect.
* If you encounter issues, consult the official documentation or community forums like Stack Overflow.

With your environment set up, you're ready to write code!

## Writing Your First F# Program (Hello World Example)

Let's create a simple "Hello World" program to introduce F# syntax and execution. This classic example demonstrates basic output and program structure.

**Step-by-Step Creation**

1. **Create a New Directory**:
   * Open a terminal and run:

mkdir HelloWorldFSharp

cd HelloWorldFSharp

* This sets up a folder for your project.

1. **Initialize the Project**:
   * Run:

dotnet new console -lang "F#"

* This generates:
  + Program.fs: The main code file.
  + HelloWorldFSharp.fsproj: The project file defining dependencies and build settings.
* Why use dotnet new? It automates boilerplate, ensuring a correct starting point with .NET integration.

1. **Edit the Code**:
   * Open Program.fs in your IDE (e.g., VS Code).
   * Replace the content with:

open System

[<EntryPoint>]

let main argv =

printfn "Hello, World from F#!"

0 // Return an integer exit code

* Save the file.

1. **Build and Run**:
   * In the terminal:

dotnet build

dotnet run

* Output: Hello, World from F#!

**In-Depth Explanation**

* **Line-by-Line Breakdown**:
  + open System: This imports the System namespace, similar to using System; in C#. It allows access to types like Console without full qualification. Namespaces organize code in .NET, preventing name clashes.
  + [<EntryPoint>]: An attribute marking the main function as the program's starting point. Attributes in F# (and .NET) provide metadata, like compiler directives. Without this, the compiler won't know where to begin execution.
  + let main argv =: Defines a function named main. let is F#'s keyword for binding values or functions. argv is an array of strings (command-line arguments), inferred by the compiler. Functions in F# are first-class, meaning they can be passed around like data.
  + printfn "Hello, World from F#!": Prints the string followed by a newline. printfn is F#'s formatted print function, safer than string concatenation due to type checking. It's inspired by functional languages like OCaml.
  + 0: Returns an exit code (0 for success). In console apps, this signals to the OS that the program ended normally.
* **Key Concepts Introduced**:
  + **Type Inference**: F# deduces types automatically. Here, main is inferred as string[] -> int without explicit declaration.
  + **Immutability by Default**: Bindings with let are immutable unless specified otherwise, promoting safer code by avoiding unintended changes.
  + **Expressions vs. Statements**: F# code consists of expressions that evaluate to values. Even printfn returns unit (like void), and the function body is an expression sequence.
* **Why This Matters**: This simple program illustrates F#'s conciseness—no unnecessary semicolons or braces. Compare to C#:

using System;

class Program {

static int Main(string[] args) {

Console.WriteLine("Hello, World from F#!");

return 0;

}

}

F# achieves the same with less code, focusing on logic over ceremony.

Experiment: Modify the string or add printfn "%s" argv.[0] (if passing arguments) to see type safety in action.

**Overview of F# Interactive (FSI) for Rapid Prototyping**

F# Interactive (FSI) is a REPL (Read-Eval-Print-Loop) tool for executing F# code snippets interactively. It's invaluable for testing ideas, exploring APIs, and learning without full project compilation.

**Getting Started with FSI**

1. **Launch FSI**:
   * In terminal: dotnet fsi
   * In VS Code with Ionide: Press Ctrl+Shift+P, type "F#: New Interactive Window", or use the Ionide panel.
   * You'll see a prompt: >
2. **Basic Usage**:
   * Type code and end with ;; to evaluate.  
     Example:

> let x = 5;;

val x: int = 5

> x \* 2;;

val it: int = 10

* val indicates a value binding. it is a special variable holding the last result.

1. **Advanced Features**:
   * Load files: #load "script.fsx";; to include external scripts.
   * Reference assemblies: #r "nuget: PackageName";; for NuGet packages (e.g., #r "nuget: FSharp.Data";;).
   * Directives like #help;; for commands.

**In-Depth Explanation**

* **Why Use FSI?**: It enables rapid prototyping—test functions or data transformations instantly without building an app. This aligns with functional programming's emphasis on pure, testable expressions.
* **Statefulness**: FSI maintains state across evaluations, so you can build incrementally (e.g., define a function, then call it later).
* **Scripting**: F# scripts (.fsx files) run in FSI, great for automation. Example script:

// myscript.fsx

printfn "Running script!"

Run with dotnet fsi myscript.fsx.

**Example: Exploring Math**:

> let square x = x \* x;;

val square: x: int -> int

> square 4;;

val it: int = 16

> [1..5] |> List.map square;;

val it: int list = [1; 4; 9; 16; 25]

Breakdown:

* let square x = x \* x: Defines a function. Type inferred as int -> int.
* [1..5]: A list literal (range).
* |>: Pipeline operator, passes the list to List.map, which applies square to each element.
* This demonstrates functional composition: chain operations declaratively.

FSI fosters experimentation, reducing the edit-compile-run cycle.

## Basic Project Structure and Tools

F# projects follow .NET conventions, ensuring interoperability.

**Key Files and Structure**

* **.fsproj File**: XML-based project file.  
  Example:

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<OutputType>Exe</OutputType>

<TargetFramework>net9.0</TargetFramework>

</PropertyGroup>

<ItemGroup>

<Compile Include="Program.fs" />

</ItemGroup>

</Project>

* <TargetFramework>: Specifies .NET version (e.g., net9.0).
* <Compile Include>: Lists .fs files to compile.
* Why XML? It's declarative and tool-agnostic.
* **.fs Files**: Source code files. Order matters—files are compiled sequentially, so dependencies must precede usage.
  + Use module to organize (e.g., module MyModule = ...).
* **Other Tools**:
  + dotnet build: Compiles the project.
  + dotnet run: Builds and executes.
  + dotnet test: For unit tests (add via dotnet new xunit -lang "F#" ).
  + dotnet add package: Adds NuGet packages (e.g., dotnet add package FSharp.Data).
* **Project Types**:
  + Console: For command-line apps.
  + Library: dotnet new classlib -lang "F#" for reusable code.
  + Web: Later chapters.

Understanding this structure prepares you for scalable development.

**Ongoing Project: Setting up the Initial Project Structure and Basic Entry Point**

Now, apply what you've learned to start our console-based task manager. This project will grow chapter by chapter.

**Project Overview (Reminder)**

We'll build a task manager for adding, viewing, and managing tasks via console input.

**Step-by-Step Setup**

1. **Create the Project**:
   * Terminal:

mkdir TaskManagerFSharp

cd TaskManagerFSharp

dotnet new console -lang "F#"

1. **Initial Code in Program.fs**:

open System

module TaskManager =

type Task = {

Description: string

IsCompleted: bool

}

let tasks = [] // Empty list of tasks for now

let printWelcome() =

printfn "Welcome to F# Task Manager!"

[<EntryPoint>]

let main argv =

TaskManager.printWelcome()

0

* Build and run: dotnet build; dotnet run
* Output: "Welcome to F# Task Manager!"

**In-Depth Explanation**

* **Module Definition**: module TaskManager = ... groups related types and functions. Modules are like static classes, organizing code without instantiation.
* **Record Type**: type Task = { Description: string; IsCompleted: bool }
  + Records are immutable data structures, perfect for data like tasks. Fields are accessed via dot notation (e.g., task.Description).
  + Why records? They provide structural equality and are concise compared to classes.
* **Value Binding**: let tasks = [] – An empty list (inferred as Task list). Lists are immutable linked lists in F#.
* **Function**: let printWelcome() = ... – A unit-returning function for output.
* **Main Function**: Calls the welcome function. In future chapters, we'll add task operations.
* **Why Start Simple?**: This establishes the structure. We'll add features like adding tasks in Chapter 3.

Experiment in FSI: Copy the module code into FSI and call TaskManager.printWelcome();;.

Congratulations! You've set up F# and started the project. In Chapter 3, we'll explore core concepts and enhance the task manager.

# 3. Core Concepts of F#

In this chapter, we delve into the foundational ideas that make F# a powerful and elegant programming language. As a functional-first language, F# draws heavily from functional programming paradigms while integrating seamlessly with the .NET ecosystem. We'll explore key principles such as functional programming basics, immutability, expressions, type inference, static typing, pattern matching, discriminated unions, and the treatment of functions as first-class citizens. Each concept will be explained in depth, with detailed reasoning, real-world analogies, and practical examples to solidify your understanding.

These core concepts form the bedrock of idiomatic F# code, emphasizing safety, conciseness, and maintainability. By the end of this chapter, you'll apply these ideas to our ongoing project by defining basic task types and simple functions for the console-based task manager.

## Functional Programming Principles in F#

Functional programming (FP) is a paradigm that treats computation as the evaluation of mathematical functions, avoiding changing state and mutable data. F# is "functional-first," meaning it encourages FP but allows other styles when needed. This approach leads to code that is easier to test, reason about, and parallelize.

## Key Principles of FP in F#

* **Pure Functions**: Functions that always produce the same output for the same input, without side effects (e.g., no modifying global variables or I/O). Purity makes functions predictable and composable.
* **Higher-Order Functions**: Functions that take other functions as arguments or return them, enabling abstraction and reuse.
* **Recursion**: Using functions that call themselves instead of loops, which aligns with mathematical induction.
* **Avoiding Mutable State**: Preferring immutable data to reduce bugs from unexpected changes.
* **Declarative Style**: Describing "what" to compute rather than "how," leading to concise code.

## Why FP in F#?

FP shines in domains like data transformation, where operations can be chained without worrying about state. For example, processing a list of numbers: in imperative code, you'd loop and mutate; in FP, you'd map and filter declaratively. F# inherits these from languages like ML and Haskell but adds .NET practicality.

Example: A pure function to double a number.

let double x = x \* 2

* Explanation: This is pure—no side effects. Calling double 5 always returns 10. Type inferred as int -> int.

Contrast with imperative (mutable) code:

let mutable y = 5

y <- y \* 2 // Mutation

FP avoids such mutations for thread-safety and clarity.

**Immutability and Expressions**

Immutability means data cannot be changed after creation, a cornerstone of FP that prevents bugs like race conditions in concurrent programs. In F#, bindings (via let) are immutable by default; use mutable sparingly.

**Benefits of Immutability**

* **Safety**: No accidental modifications.
* **Ease of Reasoning**: Code behaves like math—values are constants.
* **Performance**: Enables optimizations like sharing structures.

To mutate, explicitly declare:

let mutable counter = 0

counter <- counter + 1

But prefer immutable alternatives, like creating new values.

**Expressions vs. Statements**

F# is expression-oriented: everything evaluates to a value, even control flow. Statements (void-returning) are rare; use expressions for composability.

Example: An if-expression.

let result = if 5 > 3 then "Greater" else "Smaller"

* Breakdown: Evaluates to "Greater". No need for ternary operators—if returns a value.
* Why? Allows nesting: let msg = "Number is " + (if x > 0 then "positive" else "non-positive").

In contrast, imperative languages use statements:

// Pseudo-C#

string result;

if (5 > 3) result = "Greater"; else result = "Smaller";

F#'s way is more concise and functional.

Real-World Analogy: Immutability is like writing on stone—permanent, reliable. Mutability is like a whiteboard—flexible but erasable by anyone.

**Type Inference and Static Typing**

F# is statically typed (types checked at compile-time) but uses type inference to deduce types automatically, reducing verbosity while maintaining safety.

## Static Typing Benefits

* Catches errors early (e.g., adding string to int fails compilation).
* Enables better IDE support (IntelliSense).

**Type Inference in Action**

The compiler infers types from usage:

let add a b = a + b // Inferred as int -> int -> int

let sum = add 1 2 // sum: int

* If used with floats: add 1.0 2.0 would error unless generic.

For generics:

let identity x = x // 'a -> 'a (generic)

* 'a means any type. Calling identity 5 specializes to int -> int.

Explicit types for clarity:

let add (a: int) (b: int) : int = a + b

Why Inference? Balances safety and brevity—no redundant annotations like in Java, but safer than dynamic languages like Python.

Common Pitfall: Inference fails if ambiguous; add hints.

## Pattern Matching and Discriminated Unions

Pattern matching is a powerful control structure for deconstructing data and branching based on structure. It's like enhanced switch/case but exhaustive and type-safe.

Discriminated Unions (DUs) are sum types: values that can be one of several named cases, often with data.

**Discriminated Unions**

Define alternatives:

type Shape =

| Circle of radius: float

| Rectangle of width: float \* height: float

* Circle case holds a float; Rectangle a tuple.

Create:

let myShape = Circle 5.0

**Pattern Matching**

Match on structure:

let area shape =

match shape with

| Circle r -> System.Math.PI \* r \* r

| Rectangle (w, h) -> w \* h

* Breakdown: Deconstructs and binds (e.g., r from Circle). Exhaustive—compiler warns if cases missed.
* Why? Safer than if-else chains; handles complex data like nested unions.

Guards for conditions:

match x with

| \_ when x > 0 -> "Positive"

| 0 -> "Zero"

| \_ -> "Negative" // Wildcard \_

Analogy: Like unpacking a gift—match reveals contents and acts accordingly.

DUs model domains elegantly, e.g., type Result<'T> = Success of 'T | Failure of string for error handling without exceptions.

## Functions as First-Class Citizens

In F#, functions are values: assignable, passable, returnable—enabling higher-order programming.

**Examples**

* Assign:

let greet = fun name -> "Hello, " + name

* Partial Application (Currying):

let add a b = a + b

let addFive = add 5 // int -> int

addFive 3 // 8

* Higher-Order:

let applyTwice f x = f (f x)

applyTwice (add 1) 0 // 2

## Why First-Class?

Enables abstraction: Libraries like List.map take functions.

List.map (fun x -> x \* x) [1; 2; 3] // [1; 4; 9]

* Pipeline: [1;2;3] |> List.map (fun x -> x \* x)

Promotes reuse and modularity.

## Ongoing Project: Defining Basic Task Types and Simple Functions

Building on Chapter 2's setup, we'll apply core concepts: Use DUs for task status, pattern matching for display, immutable lists for storage, and first-class functions for operations.

**Update Program.fs**

Replace with:

open System

module TaskManager =

type Priority = Low | Medium | High

type Task = {

Description: string

Priority: Priority

IsCompleted: bool

}

let mutable tasks: Task list = [] // Mutable for simplicity; we'll make immutable later

let addTask description priority =

let newTask = { Description = description; Priority = priority; IsCompleted = false }

tasks <- newTask :: tasks

let listTasks() =

tasks |> List.iteri (fun i task ->

let status = if task.IsCompleted then "Completed" else "Pending"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

printfn "%d: %s [%s] - %s" i task.Description prioStr status)

let printWelcome() =

printfn "Welcome to F# Task Manager!"

[<EntryPoint>]

let main argv =

TaskManager.printWelcome()

TaskManager.addTask "Buy groceries" Medium

TaskManager.addTask "Finish report" High

TaskManager.listTasks()

0

* Build and run: Adds tasks and lists them.

**In-Depth Explanation**

* **DU for Priority**: type Priority = Low | Medium | High – Simple cases without data. Models choices immutably.
* **Record Update**: Task now includes Priority. Records are immutable; we create new ones.
* **Mutable List**: let mutable tasks: Task list = [] – For now; later chapters refactor to immutable.
* **Add Function**: addTask creates immutable task, prepends (::) to list (immutable operation, but list is mutable).
* **List Function**: Uses List.iteri (higher-order) to iterate with index. Pattern matches priority; if-expression for status.
* **Main**: Demonstrates usage.

Why Mutable Here? To introduce gradually; FP prefers immutable, but imperative eases entry.

Experiment in FSI: Load code, call functions.

In Chapter 4, we'll cover syntax basics and add more to the project.

# 4. F# Syntax and Basics

This chapter introduces the fundamental syntax and constructs of F#, equipping you with the tools to write and understand basic F# code. As a functional-first language, F# emphasizes clarity and conciseness, but it also supports imperative and object-oriented paradigms for flexibility. We'll cover variables and bindings, basic data types, control flow, collections (lists, sequences, arrays), and tuples and records, providing in-depth explanations, practical examples, and their application to our ongoing console-based task manager project. Each section includes detailed reasoning, analogies, and code examples to ensure a comprehensive understanding, akin to a technical textbook. By the end, you'll enhance the task manager with task storage and basic operations, solidifying your grasp of F# fundamentals.

## Variables and Bindings

In F#, variables are defined using the let keyword, which creates immutable bindings by default. This aligns with functional programming's emphasis on immutability, reducing bugs from unintended state changes.

**Understanding let Bindings**

* **Syntax**: let <name> = <expression>
* **Immutability**: Once bound, the value cannot change unless explicitly declared mutable.
* **Scope**: Bindings are scoped to their block (e.g., function or module). Inner bindings can shadow outer ones.

Example:

let x = 10

let y = x + 5 // y is 15

// x <- 20 // Error: Immutable!

* **Explanation**: x is bound to 10. y uses x in an expression. Attempting to reassign x fails because let bindings are immutable.

## Mutable Bindings

To allow changes, use mutable:

let mutable counter = 0

counter <- counter + 1 // counter is now 1

* **Assignment Operator**: <- updates mutable variables.
* **Use Case**: Useful for imperative code or stateful algorithms, but use sparingly in idiomatic F# to favor immutability.
* **Why Immutable by Default?**: Prevents accidental state changes, especially in concurrent or parallel code, making behavior predictable like mathematical constants.

## Shadowing

Instead of mutating, F# encourages shadowing to create new bindings with the same name:

let x = 10

let x = x + 5 // New binding, x is now 15

* **Why Shadow?**: Maintains immutability while allowing "updates" within a scope. The old x is inaccessible in the new scope.

Analogy: Think of let as labeling a box with a value—you can’t change the contents, but you can create a new box with the same label. Mutable bindings are like a whiteboard you can erase and rewrite.

**In Practice**

Bindings are used everywhere in F#—for values, functions, or complex types. They’re the glue holding expressions together, ensuring type-safe, predictable code.

**Basic Data Types**

F# supports a range of primitive and composite data types, leveraging .NET’s type system with functional enhancements. Types are inferred but can be explicitly specified.

**Primitive Types**

* **Integers**: int (32-bit, e.g., 42), int64 (64-bit), byte, etc.
* **Floating-Point**: float (double-precision, e.g., 3.14), decimal for high precision.
* **Boolean**: bool (true or false).
* **Character and String**: char (e.g., 'a'), string (e.g., "Hello").
* **Unit**: unit (like void, written ()), used for functions with no meaningful return.

Example:

let age: int = 30

let pi: float = 3.14159

let name: string = "Alice"

let isActive: bool = true

let letter: char = 'A'

let noValue: unit = ()

* **Type Annotations**: Optional but clarify intent or resolve ambiguity.
* **Literals**: F# supports suffixes (e.g., 42L for int64, 3.14f for float32).

## Type Inference

F# infers types from usage:

let sum = 10 + 20 // Inferred as int

let greeting = "Hello" + " World" // Inferred as string

* **Why?**: Reduces boilerplate while maintaining static typing. Errors like 5 + "hello" fail at compile-time.

**Option Type**

The option type handles absence of values safely, avoiding nulls:

let maybeNumber: int option = Some 42

let noNumber: int option = None

* **Why?**: Eliminates null reference exceptions, common in other languages. Use pattern matching to handle Some/None.

Analogy: Types are like ingredients in a recipe—each has a specific role, and F# ensures you don’t mix sugar with salt. Option is like a sealed container: it might contain a value, or it might be empty, but you always check safely.

## Control Flow

F# uses expressions for control flow, ensuring every construct returns a value. This avoids statement-heavy code, making logic composable.

**If-Then-Else**

An if expression returns a value:

let isPositive x = if x > 0 then "Positive" else "Non-positive"

* **Syntax**: if <condition> then <expr1> else <expr2>
* **Rules**: Both branches must return the same type. else is required unless the then branch returns unit.
* **Example**:

let describe x = if x > 0 then "Positive" else if x = 0 then "Zero" else "Negative"

* **Why?**: Eliminates ternary operators; nests naturally.

## Match Expressions

Pattern matching is more powerful, handling complex conditions:

let describeNumber x =

match x with

| 0 -> "Zero"

| n when n > 0 -> "Positive"

| \_ -> "Negative"

* **Components**:
  + match <expr> with: Evaluates <expr> and matches against patterns.
  + | <pattern> -> <result>: Each case specifies a pattern and result.
  + \_: Wildcard for unmatched cases.
  + when: Guards add conditions.
* **Exhaustive Checking**: Compiler ensures all cases are covered, preventing runtime errors.

Example with tuples:

let classifyPoint (x, y) =

match x, y with

| 0, 0 -> "Origin"

| x, \_ when x > 0 -> "Right half"

| \_ -> "Left half or on Y-axis"

* **Why?**: More expressive than switch/case; handles complex data.

Analogy: if is like choosing between two doors; match is like a decision tree with many branches, ensuring you cover every possibility.

## Lists, Sequences, and Arrays

F# provides three primary collection types, each optimized for different use cases.

**Lists**

Immutable, singly-linked lists, ideal for functional programming.

* **Syntax**:

let numbers = [1; 2; 3; 4] // List literal

let emptyList: int list = []

let cons = 0 :: numbers // [0; 1; 2; 3; 4]

* **Operations**:
  + :: (cons): Prepends an element.
  + List.map, List.filter, List.fold: Transform, filter, or aggregate.
* **Example**:

let squares = List.map (fun x -> x \* x) [1; 2; 3] // [1; 4; 9]

* **Performance**: O(1) for head access/prepend; O(n) for append or random access.
* **Use Case**: Sequential processing, recursion.

**Sequences**

Lazy, potentially infinite collections, computed on demand.

* **Syntax**:

let evens = seq { for i in 0..2..10 -> i } // seq [0; 2; 4; 6; 8; 10]

* **Operations**: Similar to lists (Seq.map, Seq.filter), but lazy.
* **Example**:

let firstFew = Seq.take 3 evens // seq [0; 2; 4]

* **Why Lazy?**: Saves memory for large or infinite data, like streaming.
* **Use Case**: Data pipelines, I/O.

**Arrays**

Mutable, fixed-size, contiguous memory, for performance-critical code.

* **Syntax**:

let arr = [|1; 2; 3|]

arr.[0] <- 10 // Mutable update

* **Operations**: Array.map, Array.iter, indexing (.[n]).
* **Performance**: O(1) random access; mutable for imperative style.
* **Use Case**: Numerical computations, interop with .NET.

Example Comparing:

let nums = [1; 2; 3]

let seqNums = seq { 1 .. 3 }

let arrNums = [|1; 2; 3|]

let doubled = List.map (fun x -> x \* 2) nums // [2; 4; 6]

* **Why Different Types?**: Lists for FP, sequences for lazy evaluation, arrays for performance.

Analogy: Lists are like a linked chain—immutable and sequential. Sequences are like a water stream—flowing only when needed. Arrays are like a fixed shelf—fast but rigid.

**Tuples and Records**

**Tuples**

Lightweight, immutable groups of values, often of different types.

* **Syntax**:

let point = (3, 4)

let (x, y) = point // Destructuring

* **Use Case**: Temporary grouping, function returns.
* **Example**:

let divide a b = (a / b, a % b) // Returns quotient, remainder

let quotient, remainder = divide 10 3

* **Why?**: Quick, type-safe way to bundle data without defining a type.

## Records

Immutable, named data structures with fields.

* **Syntax**:

type Person = { Name: string; Age: int }

let alice = { Name = "Alice"; Age = 30 }

* **Access**: alice.Name (dot notation).
* **Copy-and-Update**:

let olderAlice = { alice with Age = 31 }

* **Why Records?**: Structural equality, immutability, and conciseness. Better than classes for simple data.

Comparison:

let tuplePerson = ("Alice", 30)

let recordPerson = { Name = "Alice"; Age = 30 }

* Tuple: No named fields, less readable for complex data.
* Record: Named fields, better for modeling.

Analogy: Tuples are like unlabeled bags of items; records are like labeled folders, organizing data clearly.

**Ongoing Project: Implementing Task Storage Using Lists and Records**

Let’s enhance our task manager by adding task storage, retrieval, and basic operations using lists and records. We’ll refactor to use immutable lists (removing mutable), introduce control flow for user input, and leverage pattern matching for task manipulation.

**Update Program.fs**

Replace with:

open System

module TaskManager =

type Priority = Low | Medium | High

type Task = {

Id: int

Description: string

Priority: Priority

IsCompleted: bool

}

let createTask id description priority =

{ Id = id; Description = description; Priority = priority; IsCompleted = false }

let addTask task tasks =

task :: tasks

let findTask id tasks =

tasks |> List.tryFind (fun t -> t.Id = id)

let toggleComplete id tasks =

tasks |> List.map (fun t ->

if t.Id = id then { t with IsCompleted = not t.IsCompleted } else t)

let listTasks tasks =

match tasks with

| [] -> printfn "No tasks available."

| \_ ->

tasks |> List.iteri (fun i task ->

let status = if task.IsCompleted then "Completed" else "Pending"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

printfn "%d: %s [%s] - %s (ID: %d)" i task.Description prioStr status task.Id)

let rec mainLoop tasks idCounter =

printfn "\nTask Manager: [a]dd, [t]oggle complete, [l]ist, [q]uit"

let input = Console.ReadLine().ToLower()

match input with

| "a" ->

printf "Enter description: "

let desc = Console.ReadLine()

printf "Enter priority (low/medium/high): "

let prioStr = Console.ReadLine().ToLower()

let priority =

match prioStr with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium // Default

let newTask = createTask idCounter desc priority

let updatedTasks = addTask newTask tasks

mainLoop updatedTasks (idCounter + 1)

| "t" ->

printf "Enter task ID to toggle: "

let id = int (Console.ReadLine())

match findTask id tasks with

| Some task ->

printfn "Toggling task %d: %s" task.Id task.Description

mainLoop (toggleComplete id tasks) idCounter

| None ->

printfn "Task ID %d not found." id

mainLoop tasks idCounter

| "l" ->

listTasks tasks

mainLoop tasks idCounter

| "q" -> tasks

| \_ ->

printfn "Invalid input."

mainLoop tasks idCounter

[<EntryPoint>]

let main argv =

printfn "Welcome to F# Task Manager!"

let finalTasks = TaskManager.mainLoop [] 1

printfn "\nFinal task list:"

TaskManager.listTasks finalTasks

0

**In-Depth Explanation**

* **Record Update**: Added Id to Task for unique identification.
* **Immutable List**: Removed mutable tasks, using function parameters (tasks) for immutability. Each function returns a new list.
* **Functions**:
  + createTask: Builds a Task record.
  + addTask: Prepends using ::, immutable.
  + findTask: Uses List.tryFind, returning option to handle absence safely.
  + toggleComplete: Maps over tasks, updating IsCompleted for matching ID.
  + listTasks: Enhanced with index and ID display, using match for empty list check.
* **Control Flow**: mainLoop uses recursion and match for a console menu. Pattern matches user input and priority strings.
* **Recursion**: mainLoop calls itself with updated state, returning final tasks when quitting.
* **Type Safety**: Option types and matching ensure robust error handling.

**Running the Program**

* dotnet build; dotnet run
* Try: Add tasks, toggle completion, list, and quit.
* Example output:

Welcome to F# Task Manager!

Task Manager: [a]dd, [t]oggle complete, [l]ist, [q]uit

a

Enter description: Write report

Enter priority (low/medium/high): high

Task Manager: [a]dd, [t]oggle complete, [l]ist, [q]uit

l

0: Write report [High] - Pending (ID: 1)

Task Manager: [a]dd, [t]oggle complete, [l]ist, [q]uit

q

Final task list:

0: Write report [High] - Pending (ID: 1)

**Why This Approach?**

* **Immutability**: Aligns with FP, making state predictable.
* **Records**: Structure tasks clearly.
* **Lists**: Natural for task storage, with efficient prepend.
* **Match Expressions**: Robust menu handling and error checking.
* **Recursion**: Replaces imperative loops for FP purity.

Experiment in FSI: Load the module, test addTask and listTasks with sample data.

This chapter equips you with F#’s core syntax, applied practically to the task manager. Chapter 5 will explore functional programming techniques to further enhance the project.

# 5. Functional Programming in F#

This chapter dives deep into the heart of F#’s functional programming (FP) paradigm, exploring techniques that make F# code concise, expressive, and robust. We’ll cover higher-order functions, function composition, the pipeline operator, recursion with tail-call optimization, immutable data structures, and common operations like map, filter, and fold. Each concept is explained comprehensively, with detailed reasoning, practical examples, and analogies to ensure a thorough understanding, as you’d expect from a technical textbook. These techniques emphasize immutability and declarative programming, aligning with F#’s functional-first philosophy. By the end, you’ll apply these concepts to enhance our ongoing console-based task manager project, adding features like filtering and summarizing tasks. Let’s explore the power of functional programming!

## Higher-Order Functions and Function Composition

**Higher-Order Functions**

Higher-order functions (HOFs) are functions that take other functions as arguments, return functions, or both. This treats functions as first-class citizens, enabling abstraction and reuse.

* **Why HOFs?** They allow generic operations on data, like transforming or filtering collections, without repeating logic.
* **Example**: The List.map function applies a given function to each element of a list.

let double x = x \* 2

let numbers = [1; 2; 3]

let doubled = List.map double numbers // [2; 4; 6]

* **Breakdown**: double is passed to List.map, which applies it to each element. List.map has type ('a -> 'b) -> 'a list -> 'b list, where 'a -> 'b is the function argument.
* **Real-World Use**: HOFs reduce boilerplate. Instead of looping imperatively, you declare the transformation.

let names = ["Alice"; "Bob"]

let greetings = List.map (fun name -> "Hello, " + name) names // ["Hello, Alice"; "Hello, Bob"]

## Function Composition

Function composition combines two functions to create a new one, where the output of one becomes the input of another.

* **Operators**:
  + (>>) (forward composition): (f >> g) x = g (f x)
  + (<<) (backward composition): (f << g) x = f (g x)
* **Example**:

let addOne x = x + 1

let square x = x \* x

let addThenSquare = addOne >> square

let result = addThenSquare 2 // (2 + 1)^2 = 9

* **Breakdown**: addOne >> square creates a function that first adds 1, then squares the result. Type: int -> int.
* **Why?** Composition builds complex logic from simple, reusable functions, promoting modularity.
* **Contrast with Imperative**:

let result = square(addOne(2)) // Nested, less readable

Analogy: HOFs are like hiring a chef to cook any recipe you provide. Composition is like assembling a kitchen pipeline: one chef chops, another cooks, producing a dish in one flow.

## Pipelining and the Forward Operator (|>)

The pipeline operator |> passes a value through a series of functions, enhancing readability by mimicking data flow.

* **Syntax**: x |> f = f x
* **Example**:

let result = 2 |> addOne |> square // (2 + 1)^2 = 9

* **Breakdown**: 2 is passed to addOne, whose result is passed to square. Reads left-to-right, like a conveyor belt.
* **Chaining**:

let numbers = [1; 2; 3]

let result = numbers |> List.map double |> List.filter (fun x -> x > 2) // [4; 6]

* **Why?** Avoids nested calls, making code declarative and intuitive.
* **Contrast**:

let result = List.filter (fun x -> x > 2) (List.map double numbers) // Harder to follow

Analogy: Pipelining is like an assembly line: raw materials (data) pass through stations (functions) to produce a final product.

**Recursion and Tail-Call Optimization**

Recursion is a core FP technique, replacing imperative loops by having a function call itself. F# optimizes tail-recursive functions to prevent stack overflows.

**Basic Recursion**

Example: Sum a list.

let rec sumList xs =

match xs with

| [] -> 0

| head :: tail -> head + sumList tail

* **Breakdown**:
  + rec marks the function as recursive.
  + Matches on list: empty ([]) returns 0; otherwise, adds head to recursive sum of tail.
  + sumList [1; 2; 3] computes 1 + (2 + (3 + 0)) = 6.

**Tail-Call Optimization (TCO)**

Non-tail-recursive functions grow the call stack, risking overflow for large inputs. Tail-recursive functions reuse the current stack frame.

* **Non-Tail Example**:

let rec factorial n =

if n <= 1 then 1 else n \* factorial (n - 1)

* **Problem**: n \* waits for factorial (n - 1), growing the stack.
* **Tail-Recursive**:

let factorial n =

let rec loop acc n =

if n <= 1 then acc else loop (acc \* n) (n - 1)

loop 1 n

* **Breakdown**: acc accumulates the result. The recursive call is the last operation (tail position), so F# reuses the stack frame. factorial 5 computes loop 1 5 -> loop 5 4 -> loop 20 3 -> ... -> 120.
* **Why TCO?** Enables recursion for large inputs without stack overflows, critical for FP where loops are avoided.

Analogy: Recursion is like solving a puzzle by breaking it into smaller puzzles. Tail recursion is like passing a memo with the current state, avoiding a pile of papers (stack frames).

## Working with Immutable Data Structures

F#’s lists, records, and other structures are immutable by default, ensuring data integrity.

* **Lists**: Singly-linked, immutable.

let xs = [1; 2; 3]

let ys = 0 :: xs // [0; 1; 2; 3], xs unchanged

* **Records**:

type Person = { Name: string; Age: int }

let alice = { Name = "Alice"; Age = 30 }

let olderAlice = { alice with Age = 31 } // New record

* **Benefits**:
  + Thread-safety: No race conditions.
  + Predictability: No side effects.
  + Memory efficiency: Immutable structures share unchanged parts (e.g., xs and ys share [1; 2; 3]).
* **Trade-Off**: Copying for updates can seem costly, but F# optimizes via structural sharing.

Analogy: Immutable data is like a printed book—unchanging, reliable. Mutable data is like a shared notepad—prone to overwriting.

**Map, Filter, and Fold Operations**

These HOFs are staples of FP, transforming and aggregating collections declaratively.

**Map**

Applies a function to each element, producing a new collection.

let squares = List.map (fun x -> x \* x) [1; 2; 3] // [1; 4; 9]

* **Type**: ('a -> 'b) -> 'a list -> 'b list
* **Use Case**: Transform data, e.g., converting strings to uppercase.

**Filter**

Selects elements matching a predicate.

let evens = List.filter (fun x -> x % 2 = 0) [1; 2; 3; 4] // [2; 4]

* **Type**: ('a -> bool) -> 'a list -> 'a list
* **Use Case**: Subset data, e.g., active tasks.

**Fold**

Aggregates a collection into a single value using an accumulator.

let sum = List.fold (fun acc x -> acc + x) 0 [1; 2; 3] // 6

* **Type**: ('state -> 'a -> 'state) -> 'state -> 'a list -> 'state
* **Breakdown**: Starts with 0, applies + with accumulator and each element.
* **Use Case**: Summarize data, e.g., total task count.

Example Combining:

let numbers = [1; 2; 3; 4]

let result = numbers |> List.filter (fun x -> x % 2 = 0) |> List.map (fun x -> x \* x) |> List.fold (+) 0 // 20 (2^2 + 4^2)

* **Pipeline**: Filters evens, squares them, sums results.

Analogy: map is like painting each item in a row. filter is like picking apples that meet a standard. fold is like tallying scores into a final total.

## Ongoing Project: Adding Functional Operations for Task Filtering and Manipulation

We’ll enhance the task manager from Chapter 4 to use functional techniques: immutable lists, map, filter, fold, pipelining, and recursion. New features include filtering tasks by priority or status and summarizing task statistics.

**Update Program.fs**

open System

module TaskManager =

type Priority = Low | Medium | High

type Task = {

Id: int

Description: string

Priority: Priority

IsCompleted: bool

}

let createTask id description priority =

{ Id = id; Description = description; Priority = priority; IsCompleted = false }

let addTask task tasks =

task :: tasks

let findTask id tasks =

tasks |> List.tryFind (fun t -> t.Id = id)

let toggleComplete id tasks =

tasks |> List.map (fun t ->

if t.Id = id then { t with IsCompleted = not t.IsCompleted } else t)

let filterByPriority priority tasks =

tasks |> List.filter (fun t -> t.Priority = priority)

let filterByStatus isCompleted tasks =

tasks |> List.filter (fun t -> t.IsCompleted = isCompleted)

let taskSummary tasks =

let total = List.length tasks

let completed = tasks |> List.filter (fun t -> t.IsCompleted) |> List.length

let highPriority = tasks |> List.filter (fun t -> t.Priority = High) |> List.length

(total, completed, highPriority)

let listTasks tasks =

match tasks with

| [] -> printfn "No tasks available."

| \_ ->

tasks |> List.iteri (fun i task ->

let status = if task.IsCompleted then "Completed" else "Pending"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

printfn "%d: %s [%s] - %s (ID: %d)" i task.Description prioStr status task.Id)

let rec mainLoop tasks idCounter =

printfn "\nTask Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [q]uit"

let input = Console.ReadLine().ToLower()

match input with

| "a" ->

printf "Enter description: "

let desc = Console.ReadLine()

printf "Enter priority (low/medium/high): "

let prioStr = Console.ReadLine().ToLower()

let priority =

match prioStr with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

let newTask = createTask idCounter desc priority

mainLoop (addTask newTask tasks) (idCounter + 1)

| "t" ->

printf "Enter task ID to toggle: "

match Int32.TryParse(Console.ReadLine()) with

| true, id ->

match findTask id tasks with

| Some task ->

printfn "Toggling task %d: %s" task.Id task.Description

mainLoop (toggleComplete id tasks) idCounter

| None ->

printfn "Task ID %d not found." id

mainLoop tasks idCounter

| false, \_ ->

printfn "Invalid ID."

mainLoop tasks idCounter

| "p" ->

printf "Enter priority to filter (low/medium/high): "

let prioStr = Console.ReadLine().ToLower()

let priority =

match prioStr with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

let filtered = filterByPriority priority tasks

printfn "Tasks with %s priority:" prioStr

listTasks filtered

mainLoop tasks idCounter

| "s" ->

printf "Show [c]ompleted or [p]ending tasks? "

let statusStr = Console.ReadLine().ToLower()

let isCompleted = statusStr = "c"

let filtered = filterByStatus isCompleted tasks

printfn "%s tasks:" (if isCompleted then "Completed" else "Pending")

listTasks filtered

mainLoop tasks idCounter

| "m" ->

let total, completed, highPrio = taskSummary tasks

printfn "Summary: %d total tasks, %d completed, %d high-priority" total completed highPrio

mainLoop tasks idCounter

| "q" -> tasks

| \_ ->

printfn "Invalid input."

mainLoop tasks idCounter

[<EntryPoint>]

let main argv =

printfn "Welcome to F# Task Manager!"

let finalTasks = TaskManager.mainLoop [] 1

printfn "\nFinal task list:"

TaskManager.listTasks finalTasks

0

**In-Depth Explanation**

* **New Features**:
  + filterByPriority: Uses List.filter to select tasks by priority, leveraging HOFs.
  + filterByStatus: Filters by completion status, using a boolean predicate.
  + taskSummary: Uses fold-like logic (via List.length and filter) to compute statistics, returning a tuple.
* **Functional Techniques**:
  + **Pipelining**: All functions use |> for readable data flow (e.g., tasks |> List.filter ...).
  + **HOFs**: filterByPriority and filterByStatus pass predicates to List.filter.
  + **Immutability**: All operations return new lists, preserving the original tasks.
  + **Recursion**: mainLoop remains tail-recursive, ensuring efficiency.
* **Enhancements**:
  + Input validation for toggle using Int32.TryParse.
  + Menu expanded with filtering and summary options.
* **Why Functional?** Declarative operations (map, filter) make code modular and testable. Pipelining enhances clarity.

**Running the Program**

* dotnet build; dotnet run
* Try: Add tasks, filter by priority or status, view summary, toggle completion, quit.
* Example output:

Welcome to F# Task Manager!

Task Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [q]uit

a

Enter description: Write report

Enter priority (low/medium/high): high

Task Manager: ...

p

Enter priority to filter (low/medium/high): high

Tasks with high priority:

0: Write report [High] - Pending (ID: 1)

Task Manager: ...

m

Summary: 1 total tasks, 0 completed, 1 high-priority

Task Manager: ...

q

Final task list:

0: Write report [High] - Pending (ID: 1)

**Why This Approach?**

* **HOFs and Pipelining**: Make operations like filtering intuitive and reusable.
* **Immutability**: Ensures state consistency, vital for future concurrency.
* **Recursion**: Maintains FP purity, with TCO for performance.
* **Modularity**: Functions are composable, ready for further extensions.

Experiment in FSI: Test filterByPriority High [task1; task2] or taskSummary.

In Chapter 6, we’ll explore object-oriented programming in F# to add structure to the task manager, balancing FP with OOP.

# 6. Object-Oriented Programming in F#

This chapter explores F#'s object-oriented programming (OOP) capabilities, which complement its functional programming strengths. While F# is functional-first, it fully supports OOP, making it ideal for interoperating with .NET libraries, structuring complex applications, and balancing paradigms. We'll cover classes, interfaces, inheritance, records versus classes, interoperability with C# and .NET, and encapsulation with access modifiers. Each topic is explained in depth, with detailed reasoning, practical examples, and analogies to provide a textbook-level understanding. By the end, you'll apply these concepts to enhance our console-based task manager project by adding OOP elements for user interactions and task management, improving modularity and structure.

## Classes, Interfaces, and Inheritance

F# supports traditional OOP constructs like classes, interfaces, and inheritance, leveraging the .NET type system.

**Classes**

Classes in F# define objects with fields, properties, and methods. They are mutable by default but can be used immutably.

* **Syntax**:

type MyClass(name: string) =

let mutable internalState = name // Private field

member this.Name = internalState // Read-only property

member this.UpdateName(newName) = internalState <- newName // Method

* **Breakdown**:
  + type MyClass(name: string): Constructor with a parameter.
  + let mutable internalState: Private field (local binding).
  + member this.Name: Property (read-only here).
  + member this.UpdateName: Method to mutate state.
* **Instantiation**:

let obj = MyClass("Test")

printfn "%s" obj.Name // Test

obj.UpdateName("New")

printfn "%s" obj.Name // New

* **Why Classes?** Useful for stateful objects or .NET interop where classes are expected.

## Interfaces

Interfaces define contracts that classes implement, ensuring consistent behavior.

* **Syntax**:

type IPrintable =

abstract member Print : unit -> string

type Document(name: string) =

interface IPrintable with

member this.Print() = sprintf "Document: %s" name

* **Breakdown**: IPrintable declares an abstract Print method. Document implements it.
* **Usage**:

let doc = Document("Report") :> IPrintable

printfn "%s" (doc.Print()) // Document: Report

* **Upcasting (:>)**: Converts to interface type for polymorphism.
* **Why?** Enables abstraction and polymorphism, critical for .NET integration.

## Inheritance

Classes can inherit from base classes or implement multiple interfaces.

* **Syntax**:

type Animal(name: string) =

member this.Name = name

abstract member Speak : unit -> string

default this.Speak() = "Generic sound"

type Dog(name: string) =

inherit Animal(name)

override this.Speak() = "Woof!"

* **Breakdown**:
  + Animal: Base class with abstract method and default implementation.
  + Dog: Inherits, overriding Speak.
* **Usage**:

let dog = Dog("Rex")

printfn "%s says %s" dog.Name (dog.Speak()) // Rex says Woof!

* **Why?** Models hierarchies, though used less in FP due to preference for composition.

Analogy: Classes are like blueprints for physical objects (e.g., a car). Interfaces are like contracts (e.g., "must have wheels"). Inheritance is like specializing a generic car into a sports car.

## Records vs. Classes

Records and classes both structure data, but they serve different purposes.

**Records**

* **Immutable by default**, concise, with structural equality.
* **Syntax**:

type Person = { Name: string; Age: int }

* **Features**:
  + Automatic equality: { Name = "Alice"; Age = 30 } = { Name = "Alice"; Age = 30 }.
  + Copy-and-update: { person with Age = 31 }.
  + No methods, just data.
* **Use Case**: Data modeling, especially immutable data (e.g., tasks in our project).

**Classes**

* **Mutable by default**, support methods and state.
* **Syntax**:

type PersonClass(name: string, age: int) =

member this.Name = name

member this.Age = age

member this.Birthday() = PersonClass(name, age + 1)

* **Features**:
  + Reference equality (unless overridden).
  + Methods for behavior.
  + Constructor for initialization.
* **Use Case**: Stateful objects, .NET interop, or complex behavior.

**Comparison**

* **Records**: Lightweight, immutable, ideal for FP and data transfer objects.
* **Classes**: Heavier, mutable, suited for OOP patterns or .NET APIs.
* **When to Choose?** Use records for simple, immutable data; classes for stateful objects or interop.

Example:

let personRecord = { Name = "Alice"; Age = 30 }

let personClass = PersonClass("Alice", 30)

* Record is simpler; class allows methods like Birthday.

Analogy: Records are like a printed ID card—fixed data. Classes are like a robot with data and behaviors.

**Interoperability with C# and .NET Libraries**

F#’s seamless integration with .NET makes it interoperable with C# and other .NET languages.

* **Using .NET Libraries**:

open System.Collections.Generic

let list = List<string>()

list.Add("Hello")

* **Breakdown**: Uses .NET’s List<T>. F#’s open is like C#’s using.
* **Calling C# from F#**:  
  Assume a C# class:

// C# code

public class Calculator {

public int Add(int a, int b) => a + b;

}

Use in F#:

let calc = Calculator()

let sum = calc.Add(2, 3) // 5

* **Why?** F# can use any .NET library, leveraging C#’s ecosystem.
* **F# in C#**:  
  F# types (e.g., records, DUs) are compiled to .NET types, usable in C# with some caveats (e.g., DUs require pattern matching, less natural in C#).
* **Tips**:
  + Use [<CLIMutable>] on records for C# compatibility (adds setters).
  + Avoid F#-specific constructs (e.g., option) in public APIs for C#.

Analogy: F# and C# are like coworkers speaking different dialects but working on the same project (.NET), sharing tools seamlessly.

## Encapsulation and Access Modifiers

Encapsulation hides implementation details, exposing only necessary interfaces.

* **Access Modifiers**:
  + private: Visible only within the type.
  + internal: Visible within the assembly.
  + Default: public for members.
* **Example**:

type Counter() =

let mutable count = 0 // Private

member this.Increment() = count <- count + 1

member this.Value = count

* **Breakdown**: count is private; Increment and Value are public.
* **In Records**: Fields are public, but use modules for encapsulation.

module Data =

type private Record = { Value: int }

let create value = { Value = value }

* **Why?** Hides Record outside Data module.

Analogy: Encapsulation is like a vending machine—you interact via buttons (public methods), not the internal mechanics.

**Ongoing Project: Extending the Task Manager with OOP Elements**

We’ll refactor the task manager to incorporate OOP, introducing a TaskManager class to handle state and user interactions, while maintaining functional purity where possible. This balances FP and OOP, improving modularity.

**Update Program.fs**

open System

type Priority = Low | Medium | High

type Task = {

Id: int

Description: string

Priority: Priority

IsCompleted: bool

}

type ITaskManager =

abstract member AddTask : string -> Priority -> unit

abstract member ToggleComplete : int -> unit

abstract member ListTasks : unit -> unit

abstract member FilterByPriority : Priority -> unit

abstract member FilterByStatus : bool -> unit

abstract member Summary : unit -> unit

type TaskManager() =

let mutable tasks: Task list = []

let mutable idCounter = 1

let createTask description priority =

let task = { Id = idCounter; Description = description; Priority = priority; IsCompleted = false }

idCounter <- idCounter + 1

task

let printTask i task =

let status = if task.IsCompleted then "Completed" else "Pending"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

printfn "%d: %s [%s] - %s (ID: %d)" i task.Description prioStr status task.Id

interface ITaskManager with

member this.AddTask description priority =

let newTask = createTask description priority

tasks <- newTask :: tasks

member this.ToggleComplete id =

match tasks |> List.tryFind (fun t -> t.Id = id) with

| Some task ->

printfn "Toggling task %d: %s" task.Id task.Description

tasks <- tasks |> List.map (fun t ->

if t.Id = id then { t with IsCompleted = not t.IsCompleted } else t)

| None -> printfn "Task ID %d not found." id

member this.ListTasks() =

match tasks with

| [] -> printfn "No tasks available."

| \_ -> tasks |> List.iteri printTask

member this.FilterByPriority priority =

let filtered = tasks |> List.filter (fun t -> t.Priority = priority)

printfn "Tasks with %A priority:" priority

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

member this.FilterByStatus isCompleted =

let filtered = tasks |> List.filter (fun t -> t.IsCompleted = isCompleted)

printfn "%s tasks:" (if isCompleted then "Completed" else "Pending")

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

member this.Summary() =

let total = List.length tasks

let completed = tasks |> List.filter (fun t -> t.IsCompleted) |> List.length

let highPriority = tasks |> List.filter (fun t -> t.Priority = High) |> List.length

printfn "Summary: %d total tasks, %d completed, %d high-priority" total completed highPriority

member this.Run() =

let rec loop() =

printfn "\nTask Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [q]uit"

match Console.ReadLine().ToLower() with

| "a" ->

printf "Enter description: "

let desc = Console.ReadLine()

printf "Enter priority (low/medium/high): "

let priority =

match Console.ReadLine().ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

(this :> ITaskManager).AddTask desc priority

loop()

| "t" ->

printf "Enter task ID to toggle: "

match Int32.TryParse(Console.ReadLine()) with

| true, id ->

(this :> ITaskManager).ToggleComplete id

loop()

| false, \_ ->

printfn "Invalid ID."

loop()

| "p" ->

printf "Enter priority to filter (low/medium/high): "

let priority =

match Console.ReadLine().ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

(this :> ITaskManager).FilterByPriority priority

loop()

| "s" ->

printf "Show [c]ompleted or [p]ending tasks? "

let isCompleted = Console.ReadLine().ToLower() = "c"

(this :> ITaskManager).FilterByStatus isCompleted

loop()

| "m" ->

(this :> ITaskManager).Summary()

loop()

| "q" -> ()

| \_ ->

printfn "Invalid input."

loop()

printfn "Welcome to F# Task Manager!"

loop()

printfn "\nFinal task list:"

(this :> ITaskManager).ListTasks()

[<EntryPoint>]

let main argv =

let manager = TaskManager()

manager.Run()

0

**In-Depth Explanation**

* **OOP Structure**:
  + **Class**: TaskManager encapsulates state (tasks, idCounter) and behavior.
  + **Interface**: ITaskManager defines a contract for operations, enabling polymorphism and testing.
  + **Private Members**: createTask and printTask are internal helpers, encapsulated within the class.
* **State Management**:
  + Uses mutable fields for simplicity, balancing OOP and FP. Later, we could refactor to immutable patterns.
  + idCounter ensures unique task IDs, incremented in createTask.
* **Functional Integration**:
  + Retains FP techniques: List.filter, List.map, List.iteri, and pattern matching.
  + Methods like FilterByPriority use pipelines (|>) for clarity.
* **Interface Implementation**:
  + Explicit implementation via interface ... with ensures adherence to ITaskManager.
  + Upcasting (this :> ITaskManager) accesses interface methods within Run.
* **Run Method**:
  + Encapsulates the main loop, using tail-recursive loop to handle user input.
  + Centralizes interaction, improving modularity.
* **Why OOP?**:
  + Encapsulates state and behavior, making the system easier to extend (e.g., for new interfaces or subclasses).
  + Prepares for .NET interop or GUI integration.
  + Interface enables mocking for testing.

**Running the Program**

* dotnet build; dotnet run
* Example interaction:

Welcome to F# Task Manager!

Task Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [q]uit

a

Enter description: Write report

Enter priority (low/medium/high): high

Task Manager: ...

p

Enter priority to filter (low/medium/high): high

Tasks with High priority:

0: Write report [High] - Pending (ID: 1)

Task Manager: ...

q

Final task list:

0: Write report [High] - Pending (ID: 1)

**Why This Approach?**

* **Encapsulation**: TaskManager class hides implementation details, exposing only interface methods.
* **Modularity**: Interface and class structure make it easy to swap implementations (e.g., for a database-backed manager).
* **FP-OOP Balance**: Combines functional operations (e.g., List.filter) with OOP structure.
* **Extensibility**: Ready for advanced features like async or web integration.

**Experiment in FSI**

* Create a TaskManager instance:

let tm = TaskManager()

(tm :> ITaskManager).AddTask "Test" High

(tm :> ITaskManager).ListTasks()

In Chapter 7, we’ll explore advanced F# features like type providers and async workflows to further enhance the task manager, preparing for the comprehensive web project.

# 7. Advanced F# Features

This chapter delves into F#’s advanced features, which elevate its expressiveness and power for tackling complex problems. We’ll explore type providers, computation expressions, units of measure, active patterns, and quotations, providing a deep, textbook-style understanding with detailed explanations, practical examples, and analogies. These features enable seamless data integration, concise asynchronous programming, type-safe calculations, flexible pattern matching, and meta-programming. By the end, you’ll apply these concepts to enhance our console-based task manager project, adding asynchronous operations and data querying capabilities, preparing it for real-world scenarios. These advanced tools highlight F#’s unique strengths in the .NET ecosystem.

## Type Providers

Type providers are a unique F# feature that generate types at compile-time based on external data sources, such as databases, JSON, or CSV files. They provide type-safe access to data without manual schema definitions.

**How They Work**

* Type providers analyze external data (e.g., a database schema or JSON sample) and generate types dynamically.
* They integrate with IntelliSense, offering auto-completion for data fields.
* Common providers: FSharp.Data for JSON, CSV, XML; SqlClient for SQL databases.

**Example: JSON Type Provider**

Using FSharp.Data to parse JSON data.

* **Setup**: Add the NuGet package:

dotnet add package FSharp.Data

**Sample JSON** (save as sample.json):

[

{"id": 1, "name": "Alice", "score": 95},

{"id": 2, "name": "Bob", "score": 80}

]

* **Code**:

open FSharp.Data

type Scores = JsonProvider<"sample.json">

let loadScores json =

let data = Scores.Parse(json)

data |> Array.map (fun x -> x.Name, x.Score)

let jsonData = """[{"id": 3, "name": "Charlie", "score": 88}]"""

let namesAndScores = loadScores jsonData

* **Breakdown**:
  + JsonProvider<"sample.json">: Generates a type based on the JSON structure.
  + Scores.Parse: Parses runtime JSON, providing type-safe access (e.g., x.Name).
  + namesAndScores: [("Charlie", 88)].
* **Why?** Eliminates manual parsing, ensures compile-time safety.

**Why Type Providers?**

* **Productivity**: No need to write DTOs (Data Transfer Objects).
* **Safety**: Compile-time errors if data structure changes.
* **Use Case**: Accessing APIs, databases, or files with minimal boilerplate.

Analogy: A type provider is like a librarian who instantly creates a catalog for a new book collection, letting you query books (data) by title or author without manual indexing.

## Computation Expressions

Computation expressions provide a syntactic sugar for workflows like asynchronous operations, queries, or custom control flows. They make complex operations read like natural language.

**Syntax and Structure**

* **Form**:

builder { expression }

* **Common Builders**:
  + async: For asynchronous workflows.
  + seq: For lazy sequences.
  + Custom builders for domain-specific workflows.
* **Key Constructs**:
  + let!: Binds a value from a computation (e.g., async result).
  + do!: Executes a computation without binding.
  + return: Returns a value from the workflow.

**Example: Async Workflow**

open System.Threading.Tasks

let fetchDataAsync (url: string) =

async {

let! data = async { return "Simulated data" } // Simulate async fetch

return data.ToUpper()

}

let runAsync () =

let task = fetchDataAsync "http://example.com"

Async.RunSynchronously task // "SIMULATED DATA"

* **Breakdown**:
  + async { ... }: Defines an async workflow.
  + let!: Awaits the async operation, binding the result.
  + Async.RunSynchronously: Runs for demo; real apps use await in async contexts.
* **Why?** Simplifies async code, avoiding callback hell.

Analogy: Computation expressions are like a recipe book with special instructions (e.g., "wait for oven") that streamline cooking complex dishes (workflows).

## Units of Measure

Units of measure ensure type-safe calculations by associating units with numbers, preventing errors like adding meters to seconds.

**Syntax**

[<Measure>] type meter

[<Measure>] type second

let distance = 10.0<meter>

let time = 2.0<second>

let speed = distance / time // Type: float<meter/second>

* **Error Prevention**:

let invalid = distance + time // Compile-time error: mismatched units

**Example**

[<Measure>] type kg

[<Measure>] type m

[<Measure>] type s

let gravity = 9.81<m/s^2>

let mass = 5.0<kg>

let force = mass \* gravity // 49.05<kg m/s^2>

* **Breakdown**: Units are checked at compile-time, ensuring correctness.
* **Use Case**: Scientific computing, finance (e.g., currencies).

Analogy: Units are like labeled measuring cups—you can’t mix flour and water without converting, ensuring the recipe (calculation) is correct.

## Active Patterns

Active patterns allow custom pattern matching, transforming data into matchable forms.

**Syntax**

let (|PatternName|) input = transformation

* **Types**:
  + Complete: Maps input to a fixed set of cases.
  + Partial: Returns Some or None for conditional matching.

**Example: Complete Active Pattern**

let (|Even|Odd|) n = if n % 2 = 0 then Even else Odd

let describe n =

match n with

| Even -> "Even"

| Odd -> "Odd"

* **Usage**: describe 4 returns "Even".

## Partial Active Pattern

let (|Positive|\_|) n = if n > 0 then Some n else None

let checkNumber n =

match n with

| Positive x -> sprintf "Positive: %d" x

| \_ -> "Non-positive"

* **Usage**: checkNumber 5 returns "Positive: 5".
* **Why?** Simplifies complex matching logic, e.g., parsing or categorizing data.

Analogy: Active patterns are like a chef’s knife skills—slicing data into recognizable pieces for easier handling in recipes (matches).

## Quotations and Meta-Programming

Quotations represent F# code as data, enabling meta-programming (e.g., code generation or analysis).

**Syntax**

* <@ expr @>: Typed quotation, preserving type information.
* <@@ expr @@>: Untyped quotation.

**Example**

open Microsoft.FSharp.Quotations

let expr = <@ 1 + 2 @>

let evaluate expr =

match expr with

| Patterns.Call(None, op, [Patterns.Value(left, \_); Patterns.Value(right, \_)]) ->

match op.Name with

| "op\_Addition" -> left :?> int + right :?> int

| \_ -> failwith "Unknown operation"

| \_ -> failwith "Not a call"

* **Usage**: evaluate expr returns 3.
* **Breakdown**: Analyzes the expression tree, extracting and computing the sum.
* **Use Case**: Rarely used directly, but powers type providers and tools like F# CodeDom.

Analogy: Quotations are like blueprints of a building—you can inspect or modify the plans before construction.

## Ongoing Project: Enhancing the Task Manager with Async and Data Querying

We’ll enhance the task manager by adding asynchronous file I/O to save/load tasks and a type provider to query tasks from a JSON file, using FSharp.Data. This leverages async workflows and type providers, while maintaining the OOP structure from Chapter 6.

**Setup**

Add FSharp.Data:

dotnet add package FSharp.Data

**Update Program.fs**

open System

open System.IO

open FSharp.Data

open System.Threading.Tasks

[<Measure>] type day

type Priority = Low | Medium | High

type Task = {

Id: int

Description: string

Priority: Priority

IsCompleted: bool

DueDate: float<day> option

}

type TaskJson = JsonProvider<"""

[

{"id": 1, "description": "Sample task", "priority": "High", "isCompleted": false, "dueDate": 5}

]

""">

type ITaskManager =

abstract member AddTask : string -> Priority -> float<day> option -> unit

abstract member ToggleComplete : int -> unit

abstract member ListTasks : unit -> unit

abstract member FilterByPriority : Priority -> unit

abstract member FilterByStatus : bool -> unit

abstract member Summary : unit -> unit

abstract member SaveTasks : string -> unit

abstract member LoadTasks : string -> unit

type TaskManager() =

let mutable tasks: Task list = []

let mutable idCounter = 1

let createTask description priority dueDate =

let task = { Id = idCounter; Description = description; Priority = priority; IsCompleted = false; DueDate = dueDate }

idCounter <- idCounter + 1

task

let printTask i task =

let status = if task.IsCompleted then "Completed" else "Pending"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

let dueStr =

match task.DueDate with

| Some days -> sprintf "Due in %.0f days" days

| None -> "No due date"

printfn "%d: %s [%s] - %s (ID: %d, %s)" i task.Description prioStr status task.Id dueStr

let toJson (task: Task) =

sprintf """{"id": %d, "description": "%s", "priority": "%A", "isCompleted": %b, "dueDate": %s}"""

task.Id task.Description task.Priority task.IsCompleted

(match task.DueDate with Some d -> string d | None -> "null")

let (|ValidDueDate|\_|) (input: string) =

match Double.TryParse(input) with

| true, days when days >= 0.0 -> Some (Some (days \* 1.0<day>))

| \_ -> Some None

interface ITaskManager with

member this.AddTask description priority dueDate =

let newTask = createTask description priority dueDate

tasks <- newTask :: tasks

member this.ToggleComplete id =

match tasks |> List.tryFind (fun t -> t.Id = id) with

| Some task ->

printfn "Toggling task %d: %s" task.Id task.Description

tasks <- tasks |> List.map (fun t ->

if t.Id = id then { t with IsCompleted = not t.IsCompleted } else t)

| None -> printfn "Task ID %d not found." id

member this.ListTasks() =

match tasks with

| [] -> printfn "No tasks available."

| \_ -> tasks |> List.iteri printTask

member this.FilterByPriority priority =

let filtered = tasks |> List.filter (fun t -> t.Priority = priority)

printfn "Tasks with %A priority:" priority

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

member this.FilterByStatus isCompleted =

let filtered = tasks |> List.filter (fun t -> t.IsCompleted = isCompleted)

printfn "%s tasks:" (if isCompleted then "Completed" else "Pending")

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

member this.Summary() =

let total = List.length tasks

let completed = tasks |> List.filter (fun t -> t.IsCompleted) |> List.length

let highPriority = tasks |> List.filter (fun t -> t.Priority = High) |> List.length

let overdue = tasks |> List.filter (fun t ->

match t.DueDate with

| Some days -> days < 0.0<day>

| None -> false) |> List.length

printfn "Summary: %d total tasks, %d completed, %d high-priority, %d overdue" total completed highPriority overdue

member this.SaveTasks filePath =

async {

let json = sprintf "[%s]" (tasks |> List.map toJson |> String.concat ",")

do! File.WriteAllTextAsync(filePath, json) |> Async.AwaitTask

printfn "Tasks saved to %s" filePath

} |> Async.RunSynchronously

member this.LoadTasks filePath =

async {

try

let! content = File.ReadAllTextAsync(filePath) |> Async.AwaitTask

let jsonTasks = TaskJson.Parse(content)

tasks <- jsonTasks |> Array.map (fun jt ->

{ Id = jt.Id

Description = jt.Description

Priority = match jt.Priority with

| "Low" -> Low

| "Medium" -> Medium

| \_ -> High

IsCompleted = jt.IsCompleted

DueDate = if jt.DueDate.IsSome then Some (jt.DueDate.Value \* 1.0<day>) else None

}) |> Array.toList

idCounter <- (tasks |> List.map (fun t -> t.Id) |> List.max) + 1

printfn "Tasks loaded from %s" filePath

with

| ex -> printfn "Error loading tasks: %s" ex.Message

} |> Async.RunSynchronously

member this.Run() =

let rec loop() =

printfn "\nTask Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [v]save, [o]load, [q]uit"

match Console.ReadLine().ToLower() with

| "a" ->

printf "Enter description: "

let desc = Console.ReadLine()

printf "Enter priority (low/medium/high): "

let priority =

match Console.ReadLine().ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

printf "Enter due date in days (or empty for none): "

let dueDate =

match Console.ReadLine() with

| ValidDueDate d -> d

| \_ -> None

(this :> ITaskManager).AddTask desc priority dueDate

loop()

| "t" ->

printf "Enter task ID to toggle: "

match Int32.TryParse(Console.ReadLine()) with

| true, id ->

(this :> ITaskManager).ToggleComplete id

loop()

| false, \_ ->

printfn "Invalid ID."

loop()

| "p" ->

printf "Enter priority to filter (low/medium/high): "

let priority =

match Console.ReadLine().ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

(this :> ITaskManager).FilterByPriority priority

loop()

| "s" ->

printf "Show [c]ompleted or [p]ending tasks? "

let isCompleted = Console.ReadLine().ToLower() = "c"

(this :> ITaskManager).FilterByStatus isCompleted

loop()

| "m" ->

(this :> ITaskManager).Summary()

loop()

| "v" ->

printf "Enter file path to save: "

let path = Console.ReadLine()

(this :> ITaskManager).SaveTasks path

loop()

| "o" ->

printf "Enter file path to load: "

let path = Console.ReadLine()

(this :> ITaskManager).LoadTasks path

loop()

| "q" -> ()

| \_ ->

printfn "Invalid input."

loop()

printfn "Welcome to F# Task Manager!"

loop()

printfn "\nFinal task list:"

(this :> ITaskManager).ListTasks()

[<EntryPoint>]

let main argv =

let manager = TaskManager()

manager.Run()

0

**In-Depth Explanation**

* **Type Provider**:
  + TaskJson uses FSharp.Data.JsonProvider to parse tasks from JSON, ensuring type-safe access to fields like id, description, etc.
  + LoadTasks maps JSON data to Task records, handling priority and due date conversions.
  + **Why?** Simplifies file I/O, avoids manual parsing, and catches schema errors at compile-time.
* **Async Workflow**:
  + SaveTasks and LoadTasks use async for non-blocking file operations, with Async.AwaitTask to interop with .NET’s Task.
  + Runs synchronously for simplicity (Async.RunSynchronously); later chapters could make it fully async.
  + **Why?** Prepares for scalable I/O, e.g., network or database access.
* **Units of Measure**:
  + Added [<Measure>] type day and DueDate: float<day> option to Task, ensuring type-safe due date calculations.
  + Summary checks for overdue tasks (days < 0.0<day>).
  + **Why?** Prevents errors in date calculations, enhancing reliability.
* **Active Patterns**:
  + (|ValidDueDate|\_)|: Validates user input for due dates, returning Some for valid numbers or None.
  + **Why?** Simplifies input parsing in the Run loop, keeping logic clean.
* **OOP Integration**:
  + Extends ITaskManager with SaveTasks and LoadTasks.
  + Maintains encapsulation and functional operations (List.map, List.filter).
* **JSON Serialization**:
  + toJson manually formats tasks to JSON for simplicity. In production, use libraries like System.Text.Json.
  + **Why?** Demonstrates data persistence, a common requirement.

**Running the Program**

* **Setup**: Ensure FSharp.Data is installed.
* **Run**: dotnet build; dotnet run
* **Example Interaction**:

Welcome to F# Task Manager!

Task Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [v]save, [o]load, [q]uit

a

Enter description: Write report

Enter priority (low/medium/high): high

Enter due date in days (or empty for none): 3

Task Manager: ...

v

Enter file path to save: tasks.json

Tasks saved to tasks.json

Task Manager: ...

o

Enter file path to load: tasks.json

Tasks loaded from tasks.json

Task Manager: ...

m

Summary: 1 total tasks, 0 completed, 1 high-priority, 0 overdue

Task Manager: ...

q

Final task list:

0: Write report [High] - Pending (ID: 1, Due in 3 days)

* **File Output** (tasks.json):

[{"id": 1, "description": "Write report", "priority": "High", "isCompleted": false, "dueDate": 3}]

**Why This Approach?**

* **Type Providers**: Enable robust JSON handling without manual parsing.
* **Async**: Prepares for scalable I/O, critical for real-world apps.
* **Units of Measure**: Ensure correct due date handling.
* **Active Patterns**: Simplify input validation.
* **FP-OOP Balance**: Combines functional purity with OOP structure.

**Experiment in FSI**

* Test JSON loading:

let json = """[{"id": 1, "description": "Test", "priority": "High", "isCompleted": false, "dueDate": 5}]"""

let tasks = TaskJson.Parse(json)

tasks.[0].Description // "Test"

In Chapter 8, we’ll explore practical F# applications, setting the stage for the comprehensive web project in Chapter 11.

# 8. Practical F# Applications

This chapter explores practical applications of F# by applying its functional and object-oriented features to real-world scenarios. We’ll focus on data processing, scripting, unit testing, and basic web programming, providing a comprehensive, textbook-style understanding with detailed explanations, examples, and analogies. These topics showcase F#’s strengths in concise data manipulation, automation, and integration with the .NET ecosystem. By the end, you’ll enhance the console-based task manager project with unit tests and a basic web API endpoint, preparing for the comprehensive web application in Chapter 11. This chapter bridges theoretical concepts to practical, production-ready code.

## Data Processing with F#

F# excels at data processing due to its functional constructs, type safety, and type providers. It’s widely used for tasks like data transformation, analysis, and ETL (Extract, Transform, Load) processes.

**Key Features for Data Processing**

* **Type Providers**: Simplify access to structured data (e.g., CSV, JSON).
* **Functional Operations**: map, filter, fold enable declarative transformations.
* **Immutability**: Ensures reliable data pipelines.
* **Pattern Matching**: Handles complex data structures elegantly.

**Example: Processing CSV Data**

Using FSharp.Data to process a CSV file.

* **Setup**: Add the NuGet package:

dotnet add package FSharp.Data

* **Sample CSV** (save as data.csv):

Id,Name,Score

1,Alice,95

2,Bob,80

* **Code**:

open FSharp.Data

type Scores = CsvProvider<"data.csv">

let processScores filePath =

let data = Scores.Load(filePath)

data.Rows

|> Seq.filter (fun row -> row.Score > 90)

|> Seq.map (fun row -> sprintf "%s: %d" row.Name row.Score)

|> Seq.toList

* **Breakdown**:
  + CsvProvider<"data.csv">: Generates a type based on the CSV schema.
  + Scores.Load: Reads the file, providing type-safe access to columns.
  + Pipeline: Filters scores > 90, formats as strings, converts to list.
  + Example result for processScores "data.csv": ["Alice: 95"].
* **Why?** Type-safe, concise, and robust compared to manual parsing.

**Real-World Use**

* **Scenario**: Analyzing sales data or log files.
* **Benefit**: Type providers catch schema errors at compile-time; functional pipelines simplify transformations.

Analogy: Data processing in F# is like a factory conveyor belt—raw materials (data) are filtered and transformed through a series of precise, automated stations (functions).

**Scripting with F#**

F# scripts (.fsx files) are ideal for automation, prototyping, and one-off tasks, leveraging F# Interactive (FSI).

**Scripting Basics**

* **Run Scripts**: Use dotnet fsi script.fsx.
* **Directives**:
  + #load "file.fs": Include other files.
  + #r "nuget: PackageName": Reference NuGet packages.
* **Use Case**: Automating file processing, system tasks, or quick calculations.

**Example: File Cleanup Script**

#r "nuget: System.IO.Compression.FileSystem"

open System

open System.IO

let cleanOldFiles directory daysOld =

Directory.GetFiles(directory)

|> Array.filter (fun file ->

let lastWrite = File.GetLastWriteTime(file)

DateTime.Now.Subtract(lastWrite).TotalDays > daysOld)

|> Array.iter (fun file ->

File.Delete(file)

printfn "Deleted: %s" file)

cleanOldFiles @"C:\Temp" 30.0

* **Breakdown**:
  + References System.IO.Compression.FileSystem for file operations.
  + Filters files older than daysOld, deletes them, and logs.
  + Run: dotnet fsi cleanup.fsx.
* **Why?** Concise, type-safe, and reusable.

Analogy: Scripting is like a personal assistant—quickly handling repetitive tasks with clear instructions, powered by F#’s expressive syntax.

## Unit Testing in F#

Unit testing ensures code reliability, and F# integrates with .NET testing frameworks like xUnit, NUnit, or Expecto. F#’s immutable data and pure functions make testing straightforward.

**Setting Up xUnit**

* **Create Test Project**:

dotnet new xunit -lang "F#" -o TaskManagerTests

cd TaskManagerTests

dotnet add reference ../TaskManagerFSharp/TaskManagerFSharp.fsproj

* **Add xUnit Package** (if needed):

dotnet add package xunit

**Example: Testing Task Manager**

* **Test File** (Tests.fs):

open Xunit

open TaskManager

type Tests() =

[<Fact>]

let ``AddTask increases task count`` () =

let manager = TaskManager()

let initialCount = List.length manager.Tasks

(manager :> ITaskManager).AddTask "Test" Medium None

Assert.Equal(initialCount + 1, List.length manager.Tasks)

[<Fact>]

let ``ToggleComplete changes status`` () =

let manager = TaskManager()

(manager :> ITaskManager).AddTask "Test" High None

let id = 1

(manager :> ITaskManager).ToggleComplete id

let task = manager.Tasks |> List.find (fun t -> t.Id = id)

Assert.True(task.IsCompleted)

* **Breakdown**:
  + [<Fact>]: Marks a test method.
  + Tests AddTask and ToggleComplete using the TaskManager class.
  + Assert: Verifies expected behavior.
* **Run Tests**: dotnet test

**Why Test in F#?**

* **Pure Functions**: Predictable, easy to test.
* **Immutability**: Reduces side effects, simplifying test setup.
* **Type Safety**: Catches errors before runtime.

Analogy: Unit testing is like a quality control inspector—checking each component (function) to ensure the product (program) works as expected.

**Basic Web Programming with F#**

F# shines in web development, especially with frameworks like Giraffe or Suave for building APIs. We’ll introduce a basic web endpoint using Giraffe.

**Setup**

* **Create Web Project**:

dotnet new web -lang "F#" -o TaskManagerWeb

cd TaskManagerWeb

dotnet add package Giraffe

**Add Reference**:

dotnet add reference ../TaskManagerFSharp/TaskManagerFSharp.fsproj

**Example: Simple API**

* **Program.fs** (in TaskManagerWeb):

open Microsoft.AspNetCore.Builder

open Microsoft.Extensions.DependencyInjection

open Giraffe

open TaskManager

let getTasks (manager: ITaskManager) : HttpHandler =

fun next ctx ->

manager.ListTasks()

successful OK "Tasks listed in console" next ctx

let configureApp (app: IApplicationBuilder) =

let manager = TaskManager() :> ITaskManager

app.UseGiraffe (getTasks manager)

let configureServices (services: IServiceCollection) =

services.AddGiraffe() |> ignore

[<EntryPoint>]

let main argv =

let builder = WebApplication.CreateBuilder(argv)

configureServices builder.Services

let app = builder.Build()

configureApp app

app.Run()

0

* **Breakdown**:
  + Giraffe: Lightweight, functional web framework.
  + getTasks: Defines an HTTP handler that calls ListTasks and returns a response.
  + configureApp: Sets up Giraffe with the handler.
  + configureServices: Adds Giraffe services.
  + Run: dotnet run, access http://localhost:5000.

## Why Giraffe?

* Functional style: Handlers are composable functions.
* Lightweight: Minimal overhead compared to ASP.NET Core MVC.
* Interoperable: Uses ITaskManager from the main project.

Analogy: Web programming in F# is like building a storefront—Giraffe provides a clean, functional interface for customers (users) to interact with your inventory (data).

**Ongoing Project: Adding Unit Tests and a Web Endpoint**

We’ll enhance the task manager by adding unit tests to verify core functionality and a Giraffe-based API endpoint to list tasks, integrating the TaskManager class from Chapter 7.

**Step 1: Add Unit Tests**

* **Create Test Project**:

cd TaskManagerFSharp

mkdir Tests

cd Tests

dotnet new xunit -lang "F#"

dotnet add reference ../TaskManagerFSharp.fsproj

dotnet add package xunit

* **Tests.fs**:

open Xunit

open System

open TaskManager

type TaskManagerTests() =

let manager = TaskManager() :> ITaskManager

[<Fact>]

let ``AddTask adds task with correct properties`` () =

manager.AddTask "Test Task" High (Some 5.0<day>)

let task = manager.Tasks |> List.find (fun t -> t.Description = "Test Task")

Assert.Equal("Test Task", task.Description)

Assert.Equal(High, task.Priority)

Assert.Equal(Some 5.0<day>, task.DueDate)

Assert.False(task.IsCompleted)

[<Fact>]

let ``ToggleComplete updates task status`` () =

manager.AddTask "Toggle Test" Medium None

let id = manager.Tasks |> List.head |> fun t -> t.Id

manager.ToggleComplete id

let task = manager.Tasks |> List.find (fun t -> t.Id = id)

Assert.True(task.IsCompleted)

[<Fact>]

let ``FilterByPriority returns correct tasks`` () =

manager.AddTask "Low Task" Low None

let filtered = manager.Tasks |> List.filter (fun t -> t.Priority = Low)

Assert.Equal(1, filtered.Length)

Assert.Equal("Low Task", filtered.Head.Description)

[<Fact>]

let ``Save and Load preserves tasks`` () =

manager.AddTask "Save Test" Medium (Some 2.0<day>)

manager.SaveTasks "test.json"

let newManager = TaskManager() :> ITaskManager

newManager.LoadTasks "test.json"

let task = newManager.Tasks |> List.find (fun t -> t.Description = "Save Test")

Assert.Equal(Medium, task.Priority)

Assert.Equal(Some 2.0<day>, task.DueDate)

* **Breakdown**:
  + Tests AddTask, ToggleComplete, FilterByPriority, and SaveTasks/LoadTasks.
  + Uses manager.Tasks (exposing tasks field for testing, added below).
  + Run: dotnet test.

**Step 2: Modify TaskManager.fs**

Add a public Tasks property to TaskManager for testing:

open System

open System.IO

open FSharp.Data

open System.Threading.Tasks

[<Measure>] type day

type Priority = Low | Medium | High

type Task = {

Id: int

Description: string

Priority: Priority

IsCompleted: bool

DueDate: float<day> option

}

type TaskJson = JsonProvider<"""

[

{"id": 1, "description": "Sample task", "priority": "High", "isCompleted": false, "dueDate": 5}

]

""">

type ITaskManager =

abstract member AddTask : string -> Priority -> float<day> option -> unit

abstract member ToggleComplete : int -> unit

abstract member ListTasks : unit -> unit

abstract member FilterByPriority : Priority -> unit

abstract member FilterByStatus : bool -> unit

abstract member Summary : unit -> unit

abstract member SaveTasks : string -> unit

abstract member LoadTasks : string -> unit

type TaskManager() =

let mutable tasks: Task list = []

let mutable idCounter = 1

let createTask description priority dueDate =

let task = { Id = idCounter; Description = description; Priority = priority; IsCompleted = false; DueDate = dueDate }

idCounter <- idCounter + 1

task

let printTask i task =

let status = if task.IsCompleted then "Completed" else "Pending"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

let dueStr =

match task.DueDate with

| Some days -> sprintf "Due in %.0f days" days

| None -> "No due date"

printfn "%d: %s [%s] - %s (ID: %d, %s)" i task.Description prioStr status task.Id dueStr

let toJson (task: Task) =

sprintf """{"id": %d, "description": "%s", "priority": "%A", "isCompleted": %b, "dueDate": %s}"""

task.Id task.Description task.Priority task.IsCompleted

(match task.DueDate with Some d -> string d | None -> "null")

let (|ValidDueDate|\_|) (input: string) =

match Double.TryParse(input) with

| true, days when days >= 0.0 -> Some (Some (days \* 1.0<day>))

| \_ -> Some None

interface ITaskManager with

member this.AddTask description priority dueDate =

let newTask = createTask description priority dueDate

tasks <- newTask :: tasks

member this.ToggleComplete id =

match tasks |> List.tryFind (fun t -> t.Id = id) with

| Some task ->

printfn "Toggling task %d: %s" task.Id task.Description

tasks <- tasks |> List.map (fun t ->

if t.Id = id then { t with IsCompleted = not t.IsCompleted } else t)

| None -> printfn "Task ID %d not found." id

member this.ListTasks() =

match tasks with

| [] -> printfn "No tasks available."

| \_ -> tasks |> List.iteri printTask

member this.FilterByPriority priority =

let filtered = tasks |> List.filter (fun t -> t.Priority = priority)

printfn "Tasks with %A priority:" priority

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

member this.FilterByStatus isCompleted =

let filtered = tasks |> List.filter (fun t -> t.IsCompleted = isCompleted)

printfn "%s tasks:" (if isCompleted then "Completed" else "Pending")

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

member this.Summary() =

let total = List.length tasks

let completed = tasks |> List.filter (fun t -> t.IsCompleted) |> List.length

let highPriority = tasks |> List.filter (fun t -> t.Priority = High) |> List.length

let overdue = tasks |> List.filter (fun t ->

match t.DueDate with

| Some days -> days < 0.0<day>

| None -> false) |> List.length

printfn "Summary: %d total tasks, %d completed, %d high-priority, %d overdue" total completed highPriority overdue

member this.SaveTasks filePath =

async {

let json = sprintf "[%s]" (tasks |> List.map toJson |> String.concat ",")

do! File.WriteAllTextAsync(filePath, json) |> Async.AwaitTask

printfn "Tasks saved to %s" filePath

} |> Async.RunSynchronously

member this.LoadTasks filePath =

async {

try

let! content = File.ReadAllTextAsync(filePath) |> Async.AwaitTask

let jsonTasks = TaskJson.Parse(content)

tasks <- jsonTasks |> Array.map (fun jt ->

{ Id = jt.Id

Description = jt.Description

Priority = match jt.Priority with

| "Low" -> Low

| "Medium" -> Medium

| \_ -> High

IsCompleted = jt.IsCompleted

DueDate = if jt.DueDate.IsSome then Some (jt.DueDate.Value \* 1.0<day>) else None

}) |> Array.toList

idCounter <- (tasks |> List.map (fun t -> t.Id) |> List.max) + 1

printfn "Tasks loaded from %s" filePath

with

| ex -> printfn "Error loading tasks: %s" ex.Message

} |> Async.RunSynchronously

member this.Tasks = tasks // Added for testing

member this.Run() =

let rec loop() =

printfn "\nTask Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [v]save, [o]load, [q]uit"

match Console.ReadLine().ToLower() with

| "a" ->

printf "Enter description: "

let desc = Console.ReadLine()

printf "Enter priority (low/medium/high): "

let priority =

match Console.ReadLine().ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

printf "Enter due date in days (or empty for none): "

let dueDate =

match Console.ReadLine() with

| ValidDueDate d -> d

| \_ -> None

(this :> ITaskManager).AddTask desc priority dueDate

loop()

| "t" ->

printf "Enter task ID to toggle: "

match Int32.TryParse(Console.ReadLine()) with

| true, id ->

(this :> ITaskManager).ToggleComplete id

loop()

| false, \_ ->

printfn "Invalid ID."

loop()

| "p" ->

printf "Enter priority to filter (low/medium/high): "

let priority =

match Console.ReadLine().ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

(this :> ITaskManager).FilterByPriority priority

loop()

| "s" ->

printf "Show [c]ompleted or [p]ending tasks? "

let isCompleted = Console.ReadLine().ToLower() = "c"

(this :> ITaskManager).FilterByStatus isCompleted

loop()

| "m" ->

(this :> ITaskManager).Summary()

loop()

| "v" ->

printf "Enter file path to save: "

let path = Console.ReadLine()

(this :> ITaskManager).SaveTasks path

loop()

| "o" ->

printf "Enter file path to load: "

let path = Console.ReadLine()

(this :> ITaskManager).LoadTasks path

loop()

| "q" -> ()

| \_ ->

printfn "Invalid input."

loop()

printfn "Welcome to F# Task Manager!"

loop()

printfn "\nFinal task list:"

(this :> ITaskManager).ListTasks()

[<EntryPoint>]

let main argv =

let manager = TaskManager()

manager.Run()

0

**Step 3: Add Web Endpoint**

**Create Web Project**:

cd TaskManagerFSharp

mkdir TaskManagerWeb

cd TaskManagerWeb

dotnet new web -lang "F#"

dotnet add reference ../TaskManagerFSharp.fsproj

dotnet add package Giraffe

**Program.fs** (in TaskManagerWeb):

open Microsoft.AspNetCore.Builder

open Microsoft.AspNetCore.Http

open Microsoft.Extensions.DependencyInjection

open Giraffe

open TaskManager

open System.Text.Json

let getTasks (manager: ITaskManager) : HttpHandler =

fun next ctx ->

let tasks = manager.Tasks

let json = JsonSerializer.Serialize(tasks)

successful OK json next ctx

let configureApp (app: IApplicationBuilder) =

let manager = TaskManager() :> ITaskManager

app.UseGiraffe (route "/api/tasks" >=> getTasks manager)

let configureServices (services: IServiceCollection) =

services.AddGiraffe() |> ignore

[<EntryPoint>]

let main argv =

let builder = WebApplication.CreateBuilder(argv)

configureServices builder.Services

let app = builder.Build()

configureApp app

app.Run()

0

**In-Depth Explanation**

* **Unit Tests**:
  + **Structure**: Tests verify core functionality (AddTask, ToggleComplete, FilterByPriority, SaveTasks/LoadTasks).
  + **Access**: Tasks property exposes tasks for testing, balancing encapsulation with testability.
  + **Why?** Ensures reliability, especially for file I/O and data integrity.
* **Web Endpoint**:
  + **Giraffe Handler**: getTasks serializes tasks to JSON, accessible at /api/tasks.
  + **Serialization**: Uses System.Text.Json for modern JSON handling.
  + **Integration**: Reuses ITaskManager, demonstrating .NET interop.
  + **Why?** Introduces web capabilities, preparing for Chapter 11’s full-stack app.
* **Functional Elements**:
  + Retains pipelines (List.filter, List.map) within TaskManager.
  + Tests use functional constructs for assertions.
* **OOP Elements**:
  + ITaskManager ensures modularity; TaskManager encapsulates state.
* **Practicality**:
  + Tests catch regressions; API enables external access (e.g., via Postman or a browser).

## Running the Projects

* **Console App**: cd TaskManagerFSharp; dotnet run
  + Same interaction as Chapter 7, with save/load and due dates.
* **Tests**: cd Tests; dotnet test
  + Verifies functionality.
* **Web App**: cd TaskManagerWeb; dotnet run
  + Access http://localhost:5000/api/tasks to see JSON tasks:
* [{"Id":1,"Description":"Test Task","Priority":"High","IsCompleted":false,"DueDate":5}]

**Why This Approach?**

* **Testing**: Ensures robust code, critical for production.
* **Web Integration**: Lays groundwork for Chapter 11’s web app.
* **FP-OOP Balance**: Combines functional purity with OOP structure.
* **Reusability**: ITaskManager enables shared logic across console and web.

**Experiment in FSI**

* Test JSON output:

let manager = TaskManager() :> ITaskManager

manager.AddTask "Test" High (Some 5.0<day>)

System.Text.Json.JsonSerializer.Serialize(manager.Tasks)

In Chapter 9, we’ll explore concurrent and parallel programming in F# to further enhance the task manager, setting the stage for scalable applications.

# 9. Concurrent and Parallel Programming in F#

This chapter provides a comprehensive exploration of F#’s capabilities for concurrent and parallel programming, essential for building scalable, high-performance applications. We’ll cover asynchronous workflows, the Task Parallel Library (TPL), parallel programming with PSeq, mailboxes for actor-based concurrency, and thread safety with immutable data. Each topic is explained in depth, with detailed reasoning, practical examples, analogies, and considerations for performance and correctness, akin to a technical textbook. These techniques leverage F#’s functional strengths while integrating with .NET’s robust concurrency model. By the end, you’ll apply these concepts to enhance the console-based task manager project, introducing asynchronous task operations, parallel task processing, and a simple actor-based notification system, preparing for the full-stack web application in Chapter 11.

## Asynchronous Workflows

F#’s async computation expressions simplify asynchronous programming, enabling non-blocking I/O operations like file access or network requests without complex callback chains.

**Understanding Async Workflows**

* **Purpose**: Handle operations that may take time (e.g., I/O, HTTP requests) without blocking the main thread.
* **Syntax**: Uses async { ... } with let!, do!, and return to define workflows.
* **Key Operations**:
  + let!: Binds the result of an async operation.
  + do!: Executes an async operation without binding.
  + Async.RunSynchronously: Runs async code synchronously (for testing).
  + Async.AwaitTask: Interoperates with .NET Task.

**Example: Asynchronous File Read**

open System.IO

open System.Threading.Tasks

let readFileAsync path =

async {

let! content = File.ReadAllTextAsync(path) |> Async.AwaitTask

return content.ToUpper()

}

let runAsync () =

let result = readFileAsync "data.txt" |> Async.RunSynchronously

printfn "%s" result

* **Breakdown**:
  + async { ... }: Defines the workflow.
  + File.ReadAllTextAsync: .NET’s async file read, converted to F#’s async via Async.AwaitTask.
  + let!: Awaits and binds the result.
  + **Output**: Uppercase content of data.txt.

**Why Async?**

* **Non-Blocking**: Frees the thread for other tasks during I/O.
* **Composability**: async workflows chain naturally, unlike raw callbacks.
* **Error Handling**: Use try/with inside async for robust error management.

**Real-World Use**

* Fetching data from APIs, reading/writing files, or database queries.
* Example:

let fetchUrlAsync url =

async {

use client = new System.Net.Http.HttpClient()

let! response = client.GetStringAsync(url) |> Async.AwaitTask

return response

}

Analogy: Async workflows are like hiring a courier to deliver a package—you give instructions and continue other work while waiting for the delivery.

## Task Parallel Library (TPL)

The .NET TPL provides a high-level API for parallel and concurrent tasks, interoperable with F#’s async model.

**Key TPL Components**

* **Task**: Represents an asynchronous operation, similar to F#’s async but more imperative.
* **Parallel.For/ForEach**: Executes loops in parallel.
* **Task.Run**: Runs a computation on a thread pool.

**Example: Parallel Processing**

open System.Threading.Tasks

let processItems items =

items

|> Array.map (fun x -> Task.Run(fun () -> x \* x))

|> Task.WhenAll

|> Async.AwaitTask

|> Async.RunSynchronously

let squares = processItems [|1; 2; 3; 4|] // [|1; 4; 9; 16|]

* **Breakdown**:
  + Task.Run: Executes each computation (squaring) on a thread pool.
  + Task.WhenAll: Waits for all tasks to complete.
  + Async.AwaitTask: Bridges to F#’s async model.
* **Why?** Simplifies parallel execution of independent tasks.

**Considerations**

* **Thread Safety**: Ensure shared data is immutable or synchronized.
* **Overhead**: TPL is best for CPU-bound tasks, not I/O-bound (use async for I/O).

Analogy: TPL is like a team of workers in a factory—each handles a task simultaneously, coordinated to finish together.

**Parallel Programming with PSeq**

ParallelSeq (PSeq) from the FSharp.Collections.ParallelSeq library enables parallel processing of collections, ideal for CPU-bound tasks.

**Setup**

Add the NuGet package:

dotnet add package FSharp.Collections.ParallelSeq

**Example: Parallel Map**

open FSharp.Collections.ParallelSeq

let parallelSquares numbers =

numbers

|> PSeq.map (fun x -> x \* x)

|> PSeq.toArray

let result = parallelSquares [|1; 2; 3; 4|] // [|1; 4; 9; 16|]

* **Breakdown**:
  + PSeq.map: Like Seq.map, but runs in parallel on the thread pool.
  + PSeq.toArray: Collects results.
* **Why?** Simplifies parallel data processing, leveraging multi-core CPUs.

**Performance Considerations**

* **Granularity**: Ensure tasks are large enough to justify parallel overhead.
* **Data Size**: Effective for large datasets; small datasets may be slower due to coordination.
* **Immutability**: F#’s immutable collections ensure thread safety.

Analogy: PSeq is like a team of chefs chopping vegetables simultaneously—each works independently, speeding up the meal prep.

**MailboxProcessor for Actor-Based Concurrency**

MailboxProcessor implements the actor model, where agents process messages asynchronously, ideal for concurrent systems with shared state.

**How It Works**

* **Agent**: A MailboxProcessor runs a loop, receiving messages from a queue.
* **Messages**: Defined as a discriminated union or other types.
* **Concurrency**: Each agent runs on its own thread, processing messages sequentially.

**Example: Simple Counter Agent**

type CounterMsg =

| Increment of int

| Get of AsyncReplyChannel<int>

let counterAgent =

MailboxProcessor.Start(fun inbox ->

let rec loop count =

async {

let! msg = inbox.Receive()

match msg with

| Increment n -> return! loop (count + n)

| Get reply -> reply.Reply(count); return! loop count

}

loop 0)

let runCounter () =

counterAgent.Post(Increment 5)

counterAgent.Post(Increment 3)

let result = counterAgent.PostAndReply(fun reply -> Get reply)

printfn "Count: %d" result // 8

* **Breakdown**:
  + CounterMsg: Defines messages (Increment adds to count; Get retrieves it).
  + MailboxProcessor.Start: Creates an agent with a message loop.
  + Post: Sends a message asynchronously.
  + PostAndReply: Sends a message and waits for a response.
* **Why?** Safely manages state without locks, using message passing.

**Use Case**

* Concurrent systems like chat servers, event handlers, or task schedulers.

Analogy: A MailboxProcessor is like a postal worker processing letters (messages) one at a time, ensuring orderly handling without conflicts.

**Thread Safety and Immutable Data**

F#’s immutable data structures inherently support thread safety, as they cannot be modified after creation.

**Immutable Benefits**

* **No Locks Needed**: Immutable data avoids race conditions.
* **Shared Structures**: Copy-and-update (e.g., List.map) reuses unchanged parts, optimizing memory.
* **Example**:

let tasks = [{ Id = 1; Description = "Task"; Priority = Medium; IsCompleted = false; DueDate = None }]

let updated = tasks |> List.map (fun t -> { t with IsCompleted = true })

**Breakdown**: updated is a new list; tasks is unchanged, safe for concurrent access.

**Mutable Data**

When mutation is needed (e.g., in TaskManager), use synchronization:

open System.Threading

type SafeCounter() =

let mutable count = 0

let lockObj = obj()

member this.Increment() = lock lockObj (fun () -> count <- count + 1; count)

* **Why?** lock ensures atomic updates, but immutability is preferred.

Analogy: Immutable data is like a sealed letter—safe to share without tampering. Mutable data is like a shared whiteboard, requiring a lock to prevent overwriting.

**Ongoing Project: Enhancing the Task Manager with Concurrency**

We’ll enhance the task manager from Chapter 8 by:

* Making file I/O fully asynchronous using async.
* Adding parallel task processing with PSeq for summaries.
* Introducing a MailboxProcessor for task notifications.  
  This builds on the TaskManager class and prepares for the web app in Chapter 11.

**Setup**

Ensure dependencies:

cd TaskManagerFSharp

dotnet add package FSharp.Collections.ParallelSeq

**Update TaskManager.fs**

open System

open System.IO

open FSharp.Data

open System.Threading.Tasks

open FSharp.Collections.ParallelSeq

[<Measure>] type day

type Priority = Low | Medium | High

type Task = {

Id: int

Description: string

Priority: Priority

IsCompleted: bool

DueDate: float<day> option

}

type TaskJson = JsonProvider<"""

[

{"id": 1, "description": "Sample task", "priority": "High", "isCompleted": false, "dueDate": 5}

]

""">

type NotificationMsg =

| TaskAdded of Task

| TaskToggled of Task

| GetNotifications of AsyncReplyChannel<string list>

type ITaskManager =

abstract member AddTask : string -> Priority -> float<day> option -> Async<unit>

abstract member ToggleComplete : int -> Async<unit>

abstract member ListTasks : unit -> Async<unit>

abstract member FilterByPriority : Priority -> Async<unit>

abstract member FilterByStatus : bool -> Async<unit>

abstract member Summary : unit -> Async<unit>

abstract member SaveTasks : string -> Async<unit>

abstract member LoadTasks : string -> Async<unit>

abstract member GetNotifications : unit -> Async<string list>

type TaskManager() =

let mutable tasks: Task list = []

let mutable idCounter = 1

let notifier =

MailboxProcessor.Start(fun inbox ->

let rec loop notifications =

async {

let! msg = inbox.Receive()

match msg with

| TaskAdded task ->

let msg = sprintf "Added task %d: %s" task.Id task.Description

return! loop (msg :: notifications)

| TaskToggled task ->

let msg = sprintf "Toggled task %d: %s to %s" task.Id task.Description (if task.IsCompleted then "Completed" else "Pending")

return! loop (msg :: notifications)

| GetNotifications reply ->

reply.Reply(notifications)

return! loop notifications

}

loop [])

let createTask description priority dueDate =

let task = { Id = idCounter; Description = description; Priority = priority; IsCompleted = false; DueDate = dueDate }

idCounter <- idCounter + 1

task

let printTask i task =

let status = if task.IsCompleted then "Completed" else "Pending"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

let dueStr =

match task.DueDate with

| Some days -> sprintf "Due in %.0f days" days

| None -> "No due date"

printfn "%d: %s [%s] - %s (ID: %d, %s)" i task.Description prioStr status task.Id dueStr

let toJson (task: Task) =

sprintf """{"id": %d, "description": "%s", "priority": "%A", "isCompleted": %b, "dueDate": %s}"""

task.Id task.Description task.Priority task.IsCompleted

(match task.DueDate with Some d -> string d | None -> "null")

let (|ValidDueDate|\_|) (input: string) =

match Double.TryParse(input) with

| true, days when days >= 0.0 -> Some (Some (days \* 1.0<day>))

| \_ -> Some None

interface ITaskManager with

member this.AddTask description priority dueDate =

async {

let newTask = createTask description priority dueDate

tasks <- newTask :: tasks

notifier.Post(TaskAdded newTask)

}

member this.ToggleComplete id =

async {

match tasks |> List.tryFind (fun t -> t.Id = id) with

| Some task ->

printfn "Toggling task %d: %s" task.Id task.Description

tasks <- tasks |> List.map (fun t ->

if t.Id = id then { t with IsCompleted = not t.IsCompleted } else t)

notifier.Post(TaskToggled task)

| None -> printfn "Task ID %d not found." id

}

member this.ListTasks() =

async {

match tasks with

| [] -> printfn "No tasks available."

| \_ -> tasks |> List.iteri printTask

}

member this.FilterByPriority priority =

async {

let filtered = tasks |> List.filter (fun t -> t.Priority = priority)

printfn "Tasks with %A priority:" priority

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

}

member this.FilterByStatus isCompleted =

async {

let filtered = tasks |> List.filter (fun t -> t.IsCompleted = isCompleted)

printfn "%s tasks:" (if isCompleted then "Completed" else "Pending")

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

}

member this.Summary() =

async {

let total = tasks |> PSeq.length

let completed = tasks |> PSeq.filter (fun t -> t.IsCompleted) |> PSeq.length

let highPriority = tasks |> PSeq.filter (fun t -> t.Priority = High) |> PSeq.length

let overdue = tasks |> PSeq.filter (fun t ->

match t.DueDate with

| Some days -> days < 0.0<day>

| None -> false) |> PSeq.length

printfn "Summary: %d total tasks, %d completed, %d high-priority, %d overdue" total completed highPriority overdue

}

member this.SaveTasks filePath =

async {

let json = sprintf "[%s]" (tasks |> List.map toJson |> String.concat ",")

do! File.WriteAllTextAsync(filePath, json) |> Async.AwaitTask

printfn "Tasks saved to %s" filePath

}

member this.LoadTasks filePath =

async {

try

let! content = File.ReadAllTextAsync(filePath) |> Async.AwaitTask

let jsonTasks = TaskJson.Parse(content)

tasks <- jsonTasks |> Array.map (fun jt ->

{ Id = jt.Id

Description = jt.Description

Priority = match jt.Priority with

| "Low" -> Low

| "Medium" -> Medium

| \_ -> High

IsCompleted = jt.IsCompleted

DueDate = if jt.DueDate.IsSome then Some (jt.DueDate.Value \* 1.0<day>) else None

}) |> Array.toList

idCounter <- (tasks |> List.map (fun t -> t.Id) |> List.max) + 1

printfn "Tasks loaded from %s" filePath

with

| ex -> printfn "Error loading tasks: %s" ex.Message

}

member this.GetNotifications() =

async {

let! notifications = notifier.PostAndAsyncReply(fun reply -> GetNotifications reply)

return notifications

}

member this.Tasks = tasks // For testing

member this.Run() =

let rec loop() =

async {

printfn "\nTask Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [v]save, [o]load, [n]otifications, [q]uit"

let! input = Async.AwaitTask (Console.In.ReadLineAsync())

let input = input.ToLower()

match input with

| "a" ->

printf "Enter description: "

let! desc = Async.AwaitTask (Console.In.ReadLineAsync())

printf "Enter priority (low/medium/high): "

let! prioStr = Async.AwaitTask (Console.In.ReadLineAsync())

let priority =

match prioStr.ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

printf "Enter due date in days (or empty for none): "

let! dueInput = Async.AwaitTask (Console.In.ReadLineAsync())

let dueDate =

match dueInput with

| ValidDueDate d -> d

| \_ -> None

do! (this :> ITaskManager).AddTask desc priority dueDate

return! loop()

| "t" ->

printf "Enter task ID to toggle: "

let! idInput = Async.AwaitTask (Console.In.ReadLineAsync())

match Int32.TryParse(idInput) with

| true, id ->

do! (this :> ITaskManager).ToggleComplete id

return! loop()

| false, \_ ->

printfn "Invalid ID."

return! loop()

| "p" ->

printf "Enter priority to filter (low/medium/high): "

let! prioStr = Async.AwaitTask (Console.In.ReadLineAsync())

let priority =

match prioStr.ToLower() with

| "low" -> Low

| "medium" -> Medium

| "high" -> High

| \_ -> Medium

do! (this :> ITaskManager).FilterByPriority priority

return! loop()

| "s" ->

printf "Show [c]ompleted or [p]ending tasks? "

let! statusStr = Async.AwaitTask (Console.In.ReadLineAsync())

let isCompleted = statusStr.ToLower() = "c"

do! (this :> ITaskManager).FilterByStatus isCompleted

return! loop()

| "m" ->

do! (this :> ITaskManager).Summary()

return! loop()

| "v" ->

printf "Enter file path to save: "

let! path = Async.AwaitTask (Console.In.ReadLineAsync())

do! (this :> ITaskManager).SaveTasks path

return! loop()

| "o" ->

printf "Enter file path to load: "

let! path = Async.AwaitTask (Console.In.ReadLineAsync())

do! (this :> ITaskManager).LoadTasks path

return! loop()

| "n" ->

let! notifications = (this :> ITaskManager).GetNotifications()

printfn "Notifications:"

notifications |> List.iter (printfn "- %s")

return! loop()

| "q" -> ()

| \_ ->

printfn "Invalid input."

return! loop()

}

async {

printfn "Welcome to F# Task Manager!"

do! loop()

printfn "\nFinal task list:"

do! (this :> ITaskManager).ListTasks()

} |> Async.RunSynchronously

**Update Tests.fs**

Update the test project to handle async methods:

open Xunit

open System

open TaskManager

type TaskManagerTests() =

let manager = TaskManager() :> ITaskManager

[<Fact>]

let ``AddTask adds task with correct properties`` () =

async {

do! manager.AddTask "Test Task" High (Some 5.0<day>)

let task = manager.Tasks |> List.find (fun t -> t.Description = "Test Task")

Assert.Equal("Test Task", task.Description)

Assert.Equal(High, task.Priority)

Assert.Equal(Some 5.0<day>, task.DueDate)

Assert.False(task.IsCompleted)

} |> Async.RunSynchronously

[<Fact>]

let ``ToggleComplete updates status`` () =

async {

do! manager.AddTask "Toggle Test" Medium None

let id = manager.Tasks |> List.head |> fun t -> t.Id

do! manager.ToggleComplete id

let task = manager.Tasks |> List.find (fun t -> t.Id = id)

Assert.True(task.IsCompleted)

} |> Async.RunSynchronously

[<Fact>]

let ``FilterByPriority returns correct tasks`` () =

async {

do! manager.AddTask "Low Task" Low None

let filtered = manager.Tasks |> List.filter (fun t -> t.Priority = Low)

Assert.Equal(1, filtered.Length)

Assert.Equal("Low Task", filtered.Head.Description)

} |> Async.RunSynchronously

[<Fact>]

let ``Save and Load preserves tasks`` () =

async {

do! manager.AddTask "Save Test" Medium (Some 2.0<day>)

do! manager.SaveTasks "test.json"

let newManager = TaskManager() :> ITaskManager

do! newManager.LoadTasks "test.json"

let task = newManager.Tasks |> List.find (fun t -> t.Description = "Save Test")

Assert.Equal(Medium, task.Priority)

Assert.Equal(Some 2.0<day>, task.DueDate)

} |> Async.RunSynchronously

[<Fact>]

let ``Notifications record task actions`` () =

async {

do! manager.AddTask "Notify Test" High None

let id = manager.Tasks |> List.head |> fun t -> t.Id

do! manager.ToggleComplete id

let! notifications = manager.GetNotifications()

Assert.Contains("Added task", notifications |> String.concat " ")

Assert.Contains("Toggled task", notifications |> String.concat " ")

} |> Async.RunSynchronously

**In-Depth Explanation**

* **Asynchronous Workflows**:
  + **Interface Update**: ITaskManager methods return Async<unit> to support non-blocking operations.
  + **Run Method**: Fully async, using Async.AwaitTask for console input and do! for operations.
  + **Save/Load**: Already async, unchanged but integrated into the async workflow.
  + **Why?** Enables non-blocking console I/O, improving responsiveness and preparing for web integration.
* **Parallel Processing**:
  + **Summary**: Uses PSeq for parallel computation of task statistics (total, completed, highPriority, overdue).
  + **Why?** Demonstrates parallel processing, though benefits are minimal for small task lists; scalable for larger datasets.
* **MailboxProcessor**:
  + **Notifier Agent**: Tracks task additions and toggles, storing messages in a list.
  + **Messages**: TaskAdded, TaskToggled, GetNotifications for interaction.
  + **GetNotifications**: Retrieves the notification log asynchronously.
  + **Why?** Safely manages concurrent notifications, simulating real-world event logging.
* **Thread Safety**:
  + **Immutable Data**: Task operations (e.g., List.map) are thread-safe due to immutability.
  + **Mutable State**: tasks and idCounter are mutable but accessed within a single-threaded TaskManager context. The notifier agent ensures thread-safe notification handling.
* **Tests**:
  + Updated to handle Async methods, using Async.RunSynchronously for simplicity.
  + Added notification test to verify MailboxProcessor behavior.
  + **Why?** Ensures concurrent operations work correctly.

**Running the Projects**

* **Console App**:

cd TaskManagerFSharp

dotnet run

* **Example Interaction**:

Welcome to F# Task Manager!

Task Manager: [a]dd, [t]oggle, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [v]save, [o]load, [n]otifications, [q]uit

a

Enter description: Write report

Enter priority (low/medium/high): high

Enter due date in days (or empty for none): 3

Task Manager: ...

n

Notifications:

- Added task 1: Write report

Task Manager: ...

m

Summary: 1 total tasks, 0 completed, 1 high-priority, 0 overdue

Task Manager: ...

q

Final task list:

0: Write report [High] - Pending (ID: 1, Due in 3 days)

* **Tests**:

cd Tests

dotnet test

* **Web App** (from Chapter 8, unchanged): cd TaskManagerWeb; dotnet run
  + Access http://localhost:5000/api/tasks.

**Why This Approach?**

* **Async**: Ensures non-blocking I/O, critical for scalability and responsiveness.
* **Parallelism**: PSeq demonstrates CPU-bound parallel processing, extensible for large datasets.
* **Actor Model**: MailboxProcessor introduces safe, concurrent state management.
* **Thread Safety**: Immutability and agent-based concurrency minimize risks.
* **Testing**: Verifies async and concurrent behavior, ensuring reliability.

**Experiment in FSI**

* Test async and notifications:

let manager = TaskManager() :> ITaskManager

Async.RunSynchronously (manager.AddTask "Test" High (Some 5.0<day>))

Async.RunSynchronously (manager.GetNotifications()) // ["Added task 1: Test"]

**Performance Considerations**

* **Async**: Minimal overhead for I/O-bound tasks; avoids blocking threads.
* **PSeq**: Benefits large datasets but has overhead for small lists (e.g., <100 tasks).
* **MailboxProcessor**: Lightweight for message passing, but excessive agents can consume resources.
* **Optimization**: For production, profile with tools like dotnet-counters to balance parallelism and overhead.

In Chapter 10, we’ll explore domain-driven design and error handling in F#, further refining the task manager for robustness and maintainability, leading to the full-stack web app in Chapter 11.

# 10. Domain-Driven Design and Error Handling in F#

This chapter provides a comprehensive exploration of domain-driven design (DDD) and error handling in F#, focusing on techniques to model complex domains and handle errors robustly. We’ll cover DDD principles, modeling with discriminated unions and records, the Railway-Oriented Programming (ROP) pattern for error handling, and advanced pattern matching for domain logic. Each topic is explained in depth with detailed reasoning, practical examples, and analogies, delivering a textbook-level understanding. These techniques leverage F#’s type system to create expressive, type-safe, and maintainable code. By the end, you’ll refactor the console-based task manager project to incorporate DDD principles and robust error handling, ensuring a solid foundation for the full-stack web application in Chapter 11.

## Domain-Driven Design in F#

Domain-Driven Design emphasizes modeling software closely aligned with the business domain, using a ubiquitous language to ensure clarity and consistency between developers and domain experts.

**DDD Principles**

* **Ubiquitous Language**: Use domain terminology in code (e.g., Task, Priority).
* **Bounded Contexts**: Define clear boundaries for the domain model (e.g., task management vs. user management).
* **Entities and Value Objects**: Entities have identity (e.g., Task with Id); value objects are immutable without identity (e.g., Priority).
* **Aggregates**: Group related entities and value objects, with one entity (the root) controlling access.
* **Domain Events**: Model significant changes (e.g., TaskCompleted).

## F# and DDD

F#’s type system—records, discriminated unions (DUs), and immutability—makes it ideal for DDD:

* **Records**: Model entities and value objects with structural equality.
* **DUs**: Represent choices or states (e.g., Priority = Low | Medium | High).
* **Immutability**: Ensures consistent domain state.

**Example: Modeling a Task Domain**

type Priority = Low | Medium | High

type TaskId = TaskId of int

type Task = {

Id: TaskId

Description: string

Priority: Priority

Status: TaskStatus

DueDate: float<day> option

}

and TaskStatus =

| Pending

| Completed

| Cancelled

* **Breakdown**:
  + TaskId: A single-case DU to wrap int, adding type safety and preventing invalid IDs.
  + Task: A record modeling the entity, with Id as the identity.
  + TaskStatus: A DU for task states, replacing IsCompleted: bool for richer modeling.
  + **Why?** Encapsulates domain logic, ensures type safety, and aligns with ubiquitous language.

Analogy: DDD in F# is like drafting a precise blueprint for a building—every component (type) reflects the architect’s (domain expert’s) intent, ensuring clarity and correctness.

## Error Handling with Railway-Oriented Programming (ROP)

ROP is an F# idiom for handling errors functionally, treating computations as a “railway” with two tracks: success and failure.

**ROP Basics**

* **Result Type**: Use Result<'T, 'Error> to represent success (Ok 'T) or failure (Error 'Error).
* **Bind**: Chains operations, short-circuiting on failure.
* **Map**: Transforms success values without affecting failures.

**Example: Validating Input**

type DomainError =

| InvalidDescription of string

| InvalidPriority of string

| InvalidDueDate of string

let validateDescription desc =

if String.IsNullOrWhiteSpace(desc) then Error (InvalidDescription "Description cannot be empty")

else Ok desc

let validatePriority prioStr =

match prioStr.ToLower() with

| "low" -> Ok Low

| "medium" -> Ok Medium

| "high" -> Ok High

| \_ -> Error (InvalidPriority $"Invalid priority: {prioStr}")

let createTask id desc prio =

validateDescription desc

|> Result.bind (fun validDesc ->

validatePriority prio

|> Result.map (fun validPrio ->

{ Id = TaskId id; Description = validDesc; Priority = validPrio; Status = Pending; DueDate = None }))

* **Breakdown**:
  + validateDescription: Returns Ok desc or Error InvalidDescription.
  + validatePriority: Maps string to Priority or returns Error.
  + Result.bind: Chains validations, stopping at the first error.
  + Result.map: Creates the task on success.
* **Usage**:

let result = createTask 1 "" "high" // Error (InvalidDescription "Description cannot be empty")

let result2 = createTask 1 "Test" "invalid" // Error (InvalidPriority "Invalid priority: invalid")

let result3 = createTask 1 "Test" "high" // Ok { Id = TaskId 1; Description = "Test"; ... }

**Why ROP?**

* **Type Safety**: Errors are explicit in the type system, avoiding exceptions.
* **Composability**: Chains operations cleanly, handling failures gracefully.
* **Clarity**: Separates success and failure paths, improving readability.

Analogy: ROP is like a train on tracks—success keeps it on the main track, while failures switch to a sidetrack, handled without derailing the program.

## Advanced Pattern Matching

F#’s pattern matching is powerful for domain logic, especially when combined with active patterns and DUs.

**Active Patterns for Domain Logic**

let (|ValidDueDate|InvalidDueDate|) (input: string) =

match Double.TryParse(input) with

| true, days when days >= 0.0 -> ValidDueDate (Some (days \* 1.0<day>))

| \_ -> InvalidDueDate

let validateDueDate input =

match input with

| ValidDueDate days -> Ok days

| InvalidDueDate -> Error (InvalidDueDate "Invalid due date")

| "" -> Ok None

* **Breakdown**:
  + (|ValidDueDate|InvalidDueDate|): Active pattern for parsing due dates.
  + validateDueDate: Returns Result with Ok for valid dates or Error.

**Nested Pattern Matching**

let describeTask task =

match task with

| { Status = Completed; Priority = High } -> "High-priority completed task"

| { Status = Pending; DueDate = Some days } when days < 0.0<day> -> "Overdue task"

| { Status = Cancelled } -> "Cancelled task"

| \_ -> "Other task"

* **Breakdown**: Matches on multiple fields, using guards for conditions.
* **Why?** Encodes complex domain rules concisely.

Analogy: Pattern matching is like sorting mail—each letter (data) is inspected and routed to the right box (logic branch) based on its characteristics.

**Modeling Aggregates and Domain Events**

**Aggregates**

The Task entity is the aggregate root, controlling access to its state. Operations (e.g., toggling status) go through the root to maintain consistency.

**Domain Events**

Define events to capture significant changes:

type TaskEvent =

| TaskCreated of Task

| TaskStatusChanged of TaskId \* TaskStatus

| TaskCancelled of TaskId

**Why?** Events enable tracking history or triggering side effects (e.g., notifications).

**Example: Applying Events**

let applyEvent task event =

match event with

| TaskCreated newTask -> newTask

| TaskStatusChanged (TaskId id, status) when task.Id = TaskId id -> { task with Status = status }

| TaskCancelled (TaskId id) when task.Id = TaskId id -> { task with Status = Cancelled }

| \_ -> task

* **Breakdown**: Updates task state based on events, ensuring consistency.

**Ongoing Project: Refactoring Task Manager with DDD and ROP**

We’ll refactor the task manager to use DDD principles and ROP for error handling, enhancing robustness. New features include task cancellation, domain events, and comprehensive validation. The MailboxProcessor from Chapter 9 is updated to handle events.

**Update TaskManager.fs**

open System

open System.IO

open FSharp.Data

open System.Threading.Tasks

open FSharp.Collections.ParallelSeq

[<Measure>] type day

type Priority = Low | Medium | High

type TaskId = TaskId of int

type Task = {

Id: TaskId

Description: string

Priority: Priority

Status: TaskStatus

DueDate: float<day> option

}

and TaskStatus =

| Pending

| Completed

| Cancelled

type TaskEvent =

| TaskCreated of Task

| TaskStatusChanged of TaskId \* TaskStatus

| TaskCancelled of TaskId

type DomainError =

| InvalidDescription of string

| InvalidPriority of string

| InvalidDueDate of string

| TaskNotFound of int

| InvalidOperation of string

type TaskJson = JsonProvider<"""

[

{"id": 1, "description": "Sample task", "priority": "High", "status": "Pending", "dueDate": 5}

]

""">

type ITaskManager =

abstract member AddTask : string -> string -> string -> Async<Result<unit, DomainError>>

abstract member ToggleComplete : int -> Async<Result<unit, DomainError>>

abstract member CancelTask : int -> Async<Result<unit, DomainError>>

abstract member ListTasks : unit -> Async<unit>

abstract member FilterByPriority : string -> Async<Result<unit, DomainError>>

abstract member FilterByStatus : string -> Async<Result<unit, DomainError>>

abstract member Summary : unit -> Async<unit>

abstract member SaveTasks : string -> Async<Result<unit, DomainError>>

abstract member LoadTasks : string -> Async<Result<unit, DomainError>>

abstract member GetEvents : unit -> Async<TaskEvent list>

type TaskManager() =

let mutable tasks: Task list = []

let mutable idCounter = 1

let eventAgent =

MailboxProcessor.Start(fun inbox ->

let rec loop events =

async {

let! msg = inbox.Receive()

match msg with

| TaskCreated task ->

let msg = sprintf "Created task %A: %s" task.Id task.Description

return! loop (msg :: events)

| TaskStatusChanged (id, status) ->

let msg = sprintf "Changed task %A to %A" id status

return! loop (msg :: events)

| TaskCancelled id ->

let msg = sprintf "Cancelled task %A" id

return! loop (msg :: events)

| GetNotifications reply ->

reply.Reply(events)

return! loop events

}

loop [])

let createTask description priority dueDate =

let validateDescription desc =

if String.IsNullOrWhiteSpace(desc) then Error (InvalidDescription "Description cannot be empty")

else Ok desc

let validatePriority prioStr =

match prioStr.ToLower() with

| "low" -> Ok Low

| "medium" -> Ok Medium

| "high" -> Ok High

| \_ -> Error (InvalidPriority $"Invalid priority: {prioStr}")

let validateDueDate input =

match input with

| "" -> Ok None

| input ->

match Double.TryParse(input) with

| true, days when days >= 0.0 -> Ok (Some (days \* 1.0<day>))

| \_ -> Error (InvalidDueDate "Invalid due date")

validateDescription description

|> Result.bind (fun validDesc ->

validatePriority priority

|> Result.bind (fun validPrio ->

validateDueDate dueDate

|> Result.map (fun validDueDate ->

{ Id = TaskId idCounter

Description = validDesc

Priority = validPrio

Status = Pending

DueDate = validDueDate })))

let applyEvent task event =

match event with

| TaskCreated newTask -> newTask

| TaskStatusChanged (TaskId id, status) when task.Id = TaskId id -> { task with Status = status }

| TaskCancelled (TaskId id) when task.Id = TaskId id -> { task with Status = Cancelled }

| \_ -> task

let printTask i task =

let status = match task.Status with

| Pending -> "Pending"

| Completed -> "Completed"

| Cancelled -> "Cancelled"

let prioStr =

match task.Priority with

| Low -> "Low"

| Medium -> "Medium"

| High -> "High"

let dueStr =

match task.DueDate with

| Some days -> sprintf "Due in %.0f days" days

| None -> "No due date"

printfn "%d: %s [%s] - %s (ID: %A, %s)" i task.Description prioStr status task.Id dueStr

let toJson (task: Task) =

sprintf """{"id": %d, "description": "%s", "priority": "%A", "status": "%A", "dueDate": %s}"""

(let (TaskId id) = task.Id in id) task.Description task.Priority task.Status

(match task.DueDate with Some d -> string d | None -> "null")

interface ITaskManager with

member this.AddTask description priority dueDate =

async {

match createTask description priority dueDate with

| Ok newTask ->

tasks <- newTask :: tasks

eventAgent.Post(TaskCreated newTask)

return Ok ()

| Error e -> return Error e

}

member this.ToggleComplete id =

async {

match tasks |> List.tryFind (fun t -> t.Id = TaskId id) with

| Some task ->

if task.Status = Cancelled then

return Error (InvalidOperation "Cannot toggle cancelled task")

else

let newStatus = if task.Status = Completed then Pending else Completed

tasks <- tasks |> List.map (fun t -> if t.Id = TaskId id then { t with Status = newStatus } else t)

eventAgent.Post(TaskStatusChanged (TaskId id, newStatus))

return Ok ()

| None -> return Error (TaskNotFound id)

}

member this.CancelTask id =

async {

match tasks |> List.tryFind (fun t -> t.Id = TaskId id) with

| Some task ->

if task.Status = Cancelled then

return Error (InvalidOperation "Task already cancelled")

else

tasks <- tasks |> List.map (fun t -> if t.Id = TaskId id then { t with Status = Cancelled } else t)

eventAgent.Post(TaskCancelled (TaskId id))

return Ok ()

| None -> return Error (TaskNotFound id)

}

member this.ListTasks() =

async {

match tasks with

| [] -> printfn "No tasks available."

| \_ -> tasks |> List.iteri printTask

}

member this.FilterByPriority priority =

async {

match priority.ToLower() with

| "low" -> let filtered = tasks |> List.filter (fun t -> t.Priority = Low)

printfn "Tasks with Low priority:"

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

return Ok ()

| "medium" -> let filtered = tasks |> List.filter (fun t -> t.Priority = Medium)

printfn "Tasks with Medium priority:"

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

return Ok ()

| "high" -> let filtered = tasks |> List.filter (fun t -> t.Priority = High)

printfn "Tasks with High priority:"

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

return Ok ()

| \_ -> return Error (InvalidPriority $"Invalid priority: {priority}")

}

member this.FilterByStatus status =

async {

match status.ToLower() with

| "pending" -> let filtered = tasks |> List.filter (fun t -> t.Status = Pending)

printfn "Pending tasks:"

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

return Ok ()

| "completed" -> let filtered = tasks |> List.filter (fun t -> t.Status = Completed)

printfn "Completed tasks:"

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

return Ok ()

| "cancelled" -> let filtered = tasks |> List.filter (fun t -> t.Status = Cancelled)

printfn "Cancelled tasks:"

match filtered with

| [] -> printfn "No tasks found."

| \_ -> filtered |> List.iteri printTask

return Ok ()

| \_ -> return Error (InvalidOperation $"Invalid status: {status}")

}

member this.Summary() =

async {

let total = tasks |> PSeq.length

let completed = tasks |> PSeq.filter (fun t -> t.Status = Completed) |> PSeq.length

let highPriority = tasks |> PSeq.filter (fun t -> t.Priority = High) |> PSeq.length

let overdue = tasks |> PSeq.filter (fun t ->

match t.DueDate with

| Some days -> days < 0.0<day>

| None -> false) |> PSeq.length

printfn "Summary: %d total tasks, %d completed, %d high-priority, %d overdue" total completed highPriority overdue

}

member this.SaveTasks filePath =

async {

try

let json = sprintf "[%s]" (tasks |> List.map toJson |> String.concat ",")

do! File.WriteAllTextAsync(filePath, json) |> Async.AwaitTask

printfn "Tasks saved to %s" filePath

return Ok ()

with

| ex -> return Error (InvalidOperation $"Error saving tasks: {ex.Message}")

}

member this.LoadTasks filePath =

async {

try

let! content = File.ReadAllTextAsync(filePath) |> Async.AwaitTask

let jsonTasks = TaskJson.Parse(content)

tasks <- jsonTasks |> Array.map (fun jt ->

{ Id = TaskId jt.Id

Description = jt.Description

Priority = match jt.Priority with

| "Low" -> Low

| "Medium" -> Medium

| \_ -> High

Status = match jt.Status with

| "Pending" -> Pending

| "Completed" -> Completed

| \_ -> Cancelled

DueDate = if jt.DueDate.IsSome then Some (jt.DueDate.Value \* 1.0<day>) else None

}) |> Array.toList

idCounter <- (tasks |> List.map (fun t -> let (TaskId id) = t.Id in id) |> List.max) + 1

printfn "Tasks loaded from %s" filePath

return Ok ()

with

| ex -> return Error (InvalidOperation $"Error loading tasks: {ex.Message}")

}

member this.GetEvents() =

async {

let! events = eventAgent.PostAndAsyncReply(fun reply -> GetNotifications reply)

return events

}

member this.Tasks = tasks // For testing

member this.Run() =

let rec loop() =

async {

printfn "\nTask Manager: [a]dd, [t]oggle, [c]ancel, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [v]save, [o]load, [e]vents, [q]uit"

let! input = Async.AwaitTask (Console.In.ReadLineAsync())

let input = input.ToLower()

match input with

| "a" ->

printf "Enter description: "

let! desc = Async.AwaitTask (Console.In.ReadLineAsync())

printf "Enter priority (low/medium/high): "

let! prioStr = Async.AwaitTask (Console.In.ReadLineAsync())

printf "Enter due date in days (or empty for none): "

let! dueInput = Async.AwaitTask (Console.In.ReadLineAsync())

let! result = (this :> ITaskManager).AddTask desc prioStr dueInput

match result with

| Ok () -> ()

| Error e -> printfn "Error: %A" e

return! loop()

| "t" ->

printf "Enter task ID to toggle: "

let! idInput = Async.AwaitTask (Console.In.ReadLineAsync())

match Int32.TryParse(idInput) with

| true, id ->

let! result = (this :> ITaskManager).ToggleComplete id

match result with

| Ok () -> ()

| Error e -> printfn "Error: %A" e

return! loop()

| false, \_ ->

printfn "Invalid ID."

return! loop()

| "c" ->

printf "Enter task ID to cancel: "

let! idInput = Async.AwaitTask (Console.In.ReadLineAsync())

match Int32.TryParse(idInput) with

| true, id ->

let! result = (this :> ITaskManager).CancelTask id

match result with

| Ok () -> ()

| Error e -> printfn "Error: %A" e

return! loop()

| false, \_ ->

printfn "Invalid ID."

return! loop()

| "p" ->

printf "Enter priority to filter (low/medium/high): "

let! prioStr = Async.AwaitTask (Console.In.ReadLineAsync())

let! result = (this :> ITaskManager).FilterByPriority prioStr

match result with

| Ok () -> ()

| Error e -> printfn "Error: %A" e

return! loop()

| "s" ->

printf "Enter status to filter (pending/completed/cancelled): "

let! statusStr = Async.AwaitTask (Console.In.ReadLineAsync())

let! result = (this :> ITaskManager).FilterByStatus statusStr

match result with

| Ok () -> ()

| Error e -> printfn "Error: %A" e

return! loop()

| "m" ->

do! (this :> ITaskManager).Summary()

return! loop()

| "v" ->

printf "Enter file path to save: "

let! path = Async.AwaitTask (Console.In.ReadLineAsync())

let! result = (this :> ITaskManager).SaveTasks path

match result with

| Ok () -> ()

| Error e -> printfn "Error: %A" e

return! loop()

| "o" ->

printf "Enter file path to load: "

let! path = Async.AwaitTask (Console.In.ReadLineAsync())

let! result = (this :> ITaskManager).LoadTasks path

match result with

| Ok () -> ()

| Error e -> printfn "Error: %A" e

return! loop()

| "e" ->

let! events = (this :> ITaskManager).GetEvents()

printfn "Events:"

events |> List.iter (printfn "- %s")

return! loop()

| "q" -> ()

| \_ ->

printfn "Invalid input."

return! loop()

}

async {

printfn "Welcome to F# Task Manager!"

do! loop()

printfn "\nFinal task list:"

do! (this :> ITaskManager).ListTasks()

} |> Async.RunSynchronously

**Update Tests.fs**

Update tests to handle Result and new domain model:

open Xunit

open System

open TaskManager

type TaskManagerTests() =

let manager = TaskManager() :> ITaskManager

[<Fact>]

let ``AddTask adds valid task`` () =

async {

let! result = manager.AddTask "Test Task" "High" "5"

match result with

| Ok () ->

let task = manager.Tasks |> List.find (fun t -> t.Description = "Test Task")

Assert.Equal("Test Task", task.Description)

Assert.Equal(High, task.Priority)

Assert.Equal(Some 5.0<day>, task.DueDate)

Assert.Equal(Pending, task.Status)

| Error e -> Assert.Fail($"Expected Ok, got {e}")

} |> Async.RunSynchronously

[<Fact>]

let ``AddTask fails on empty description`` () =

async {

let! result = manager.AddTask "" "Medium" ""

match result with

| Error (InvalidDescription \_) -> ()

| \_ -> Assert.Fail("Expected InvalidDescription error")

} |> Async.RunSynchronously

[<Fact>]

let ``ToggleComplete updates status`` () =

async {

do! manager.AddTask "Toggle Test" "Medium" "" |> Async.Ignore

let id = (manager.Tasks |> List.head).Id |> fun (TaskId id) -> id

let! result = manager.ToggleComplete id

match result with

| Ok () ->

let task = manager.Tasks |> List.find (fun t -> t.Id = TaskId id)

Assert.Equal(Completed, task.Status)

| Error e -> Assert.Fail($"Expected Ok, got {e}")

} |> Async.RunSynchronously

[<Fact>]

let ``CancelTask updates status`` () =

async {

do! manager.AddTask "Cancel Test" "Low" "" |> Async.Ignore

let id = (manager.Tasks |> List.head).Id |> fun (TaskId id) -> id

let! result = manager.CancelTask id

match result with

| Ok () ->

let task = manager.Tasks |> List.find (fun t -> t.Id = TaskId id)

Assert.Equal(Cancelled, task.Status)

| Error e -> Assert.Fail($"Expected Ok, got {e}")

} |> Async.RunSynchronously

[<Fact>]

let ``Events record task actions`` () =

async {

do! manager.AddTask "Event Test" "High" "" |> Async.Ignore

let id = (manager.Tasks |> List.head).Id |> fun (TaskId id) -> id

do! manager.ToggleComplete id |> Async.Ignore

let! events = manager.GetEvents()

Assert.Contains("Created task", events |> String.concat " ")

Assert.Contains("Changed task", events |> String.concat " ")

} |> Async.RunSynchronously

**In-Depth Explanation**

* **DDD Implementation**:
  + **TaskId**: Single-case DU ensures type-safe IDs, preventing invalid values.
  + **TaskStatus**: DU replaces IsCompleted for richer state modeling (Pending, Completed, Cancelled).
  + **Task**: Aggregate root, controlling state changes via operations like ToggleComplete and CancelTask.
  + **TaskEvent**: Captures domain events for auditing and side effects.
  + **Why?** Aligns code with domain concepts, enhancing maintainability and clarity.
* **ROP Error Handling**:
  + **DomainError**: DU for errors (e.g., InvalidDescription, TaskNotFound).
  + **Validation**: createTask uses ROP to chain validations, short-circuiting on failure.
  + **Interface**: Methods return Async<Result<unit, DomainError>>, ensuring errors are explicit.
  + **Why?** Avoids exceptions, makes error paths clear, and integrates with async workflows.
* **Advanced Pattern Matching**:
  + createTask uses Result.bind and Result.map for validation.
  + applyEvent leverages pattern matching for event-driven updates.
  + **Why?** Encodes domain rules concisely and safely.
* **Event Handling**:
  + eventAgent (updated MailboxProcessor) logs events as strings for simplicity.
  + TaskCreated, TaskStatusChanged, TaskCancelled capture domain changes.
  + **Why?** Enables auditing and extensibility (e.g., for notifications).
* **Async Integration**:
  + All operations remain async, using Async<Result<...>> to combine asynchrony with error handling.
  + Run method handles Result outcomes, displaying errors to the user.
* **Tests**:
  + Updated to test Result-based APIs and new cancellation feature.
  + Verify domain constraints (e.g., empty description fails).
  + **Why?** Ensures domain rules and error handling work as expected.

**Running the Projects**

* **Console App**:

cd TaskManagerFSharp

dotnet run

* **Example Interaction**:

Welcome to F# Task Manager!

Task Manager: [a]dd, [t]oggle, [c]ancel, [l]ist, [p]riority filter, [s]tatus filter, [m]summary, [v]save, [o]load, [e]vents, [q]uit

a

Enter description: Write report

Enter priority (low/medium/high): high

Enter due date in days (or empty for none): 3

Task Manager: ...

c

Enter task ID to cancel: 1

Task Manager: ...

e

Events:

- Created task TaskId 1: Write report

- Cancelled task TaskId 1

Task Manager: ...

l

0: Write report [High] - Cancelled (ID: TaskId 1, Due in 3 days)

Task Manager: ...

q

Final task list:

0: Write report [High] - Cancelled (ID: TaskId 1, Due in 3 days)

* **Tests**:

cd Tests

dotnet test

* **Web App** (unchanged from Chapter 8): cd TaskManagerWeb; dotnet run
  + Note: Web app needs updates to handle new domain model in Chapter 11.

**Why This Approach?**

* **DDD**: TaskId, TaskStatus, and TaskEvent align code with the task management domain, improving clarity and extensibility.
* **ROP**: Explicit error handling avoids surprises, integrates with async, and supports testing.
* **Pattern Matching**: Simplifies complex domain logic, ensuring correctness.
* **Events**: Enable auditing and future extensions (e.g., email notifications).
* **Scalability**: Prepares for Chapter 11’s web app by providing a robust domain model.

**Experiment in FSI**

* Test task creation and events:

let manager = TaskManager() :> ITaskManager

Async.RunSynchronously (manager.AddTask "Test" "High" "5") // Ok ()

Async.RunSynchronously (manager.GetEvents()) // ["Created task TaskId 1: Test"]

In Chapter 11, we’ll build a full-stack web application, integrating the refactored task manager with a Giraffe-based API and a front-end, leveraging the robust domain model and error handling established here.

# 11. Building a Full-Stack Web Application in F#

This chapter culminates the F# learning journey by building a full-stack web application for the task manager, integrating the robust domain model and error handling from Chapter 10. We’ll use **Giraffe** for the backend API, **React** with **Tailwind CSS** for the frontend, and enhance the backend with asynchronous operations, domain-driven design (DDD), and Railway-Oriented Programming (ROP). This comprehensive, textbook-style exploration covers API development, frontend integration, CORS configuration, and deployment considerations, with detailed explanations, practical examples, and analogies. By the end, you’ll have a fully functional task management web app, demonstrating F#’s power in modern web development and its seamless .NET integration.

## Overview of the Full-Stack Application

The task manager web app allows users to:

* Add, toggle, cancel, and list tasks.
* Filter tasks by priority or status.
* View summaries and event logs.
* Persist tasks to a JSON file.

**Architecture**

* **Backend**: F# with Giraffe, exposing a RESTful API.
* **Frontend**: React with Tailwind CSS, hosted in a single-page application (SPA).
* **Data Storage**: JSON file for simplicity (extensible to a database).
* **Features**: Asynchronous operations, DDD, ROP, and event logging via MailboxProcessor.

**Tools and Setup**

* **Backend Dependencies**:
  + Giraffe: Functional web framework.
  + FSharp.Data: For JSON type provider.
  + FSharp.Collections.ParallelSeq: For parallel processing.
* **Frontend Dependencies**:
  + React: UI library via CDN.
  + Tailwind CSS: Styling via CDN.
* **Project Structure**:
  + TaskManagerFSharp: Backend logic (updated from Chapter 10).
  + TaskManagerWeb: Web app hosting API and frontend.
  + Tests: Unit tests (updated for API).

**Backend: Enhancing the Task Manager API**

We’ll extend the TaskManager from Chapter 10, adding a RESTful API with Giraffe. The API will handle CRUD operations and integrate with the existing DDD and ROP model.

**Update TaskManager.fs**

This refines the Chapter 10 code, keeping the domain model and adding API-friendly methods.

open System

open System.IO

open FSharp.Data

open System.Threading.Tasks

open FSharp.Collections.ParallelSeq

[<Measure>] type day

type Priority = Low | Medium | High

type TaskId = TaskId of int

type Task = {

Id: TaskId

Description: string

Priority: Priority

Status: TaskStatus

DueDate: float<day> option

}

and TaskStatus =

| Pending

| Completed

| Cancelled

type TaskEvent =

| TaskCreated of Task

| TaskStatusChanged of TaskId \* TaskStatus

| TaskCancelled of TaskId

type DomainError =

| InvalidDescription of string

| InvalidPriority of string

| InvalidDueDate of string

| TaskNotFound of int

| InvalidOperation of string

type TaskJson = JsonProvider<"""

[

{"id": 1, "description": "Sample task", "priority": "High", "status": "Pending", "dueDate": 5}

]

""">

type ITaskManager =

abstract member AddTask : string -> string -> string -> Async<Result<unit, DomainError>>

abstract member ToggleComplete : int -> Async<Result<unit, DomainError>>

abstract member CancelTask : int -> Async<Result<unit, DomainError>>

abstract member ListTasks : unit -> Async<Task list>

abstract member FilterByPriority : string -> Async<Result<Task list, DomainError>>

abstract member FilterByStatus : string -> Async<Result<Task list, DomainError>>

abstract member Summary : unit -> Async<(int \* int \* int \* int)>

abstract member SaveTasks : string -> Async<Result<unit, DomainError>>

abstract member LoadTasks : string -> Async<Result<unit, DomainError>>

abstract member GetEvents : unit -> Async<string list>

type TaskManager() =

let mutable tasks: Task list = []

let mutable idCounter = 1

let eventAgent =

MailboxProcessor.Start(fun inbox ->

let rec loop events =

async {

let! msg = inbox.Receive()

match msg with

| TaskCreated task ->

let msg = sprintf "Created task %A: %s" task.Id task.Description

return! loop (msg :: events)

| TaskStatusChanged (id, status) ->

let msg = sprintf "Changed task %A to %A" id status

return! loop (msg :: events)

| TaskCancelled id ->

let msg = sprintf "Cancelled task %A" id

return! loop (msg :: events)

| GetNotifications reply ->

reply.Reply(events)

return! loop events

}

loop [])

let createTask description priority dueDate =

let validateDescription desc =

if String.IsNullOrWhiteSpace(desc) then Error (InvalidDescription "Description cannot be empty")

else Ok desc

let validatePriority prioStr =

match prioStr.ToLower() with

| "low" -> Ok Low

| "medium" -> Ok Medium

| "high" -> Ok High

| \_ -> Error (InvalidPriority $"Invalid priority: {prioStr}")

let validateDueDate input =

match input with

| "" -> Ok None

| input ->

match Double.TryParse(input) with

| true, days when days >= 0.0 -> Ok (Some (days \* 1.0<day>))

| \_ -> Error (InvalidDueDate "Invalid due date")

validateDescription description

|> Result.bind (fun validDesc ->

validatePriority priority

|> Result.bind (fun validPrio ->

validateDueDate dueDate

|> Result.map (fun validDueDate ->

{ Id = TaskId idCounter

Description = validDesc

Priority = validPrio

Status = Pending

DueDate = validDueDate })))

let toJson (task: Task) =

sprintf """{"id": %d, "description": "%s", "priority": "%A", "status": "%A", "dueDate": %s}"""

(let (TaskId id) = task.Id in id) task.Description task.Priority task.Status

(match task.DueDate with Some d -> string d | None -> "null")

interface ITaskManager with

member this.AddTask description priority dueDate =

async {

match createTask description priority dueDate with

| Ok newTask ->

tasks <- newTask :: tasks

idCounter <- idCounter + 1

eventAgent.Post(TaskCreated newTask)

return Ok ()

| Error e -> return Error e

}

member this.ToggleComplete id =

async {

match tasks |> List.tryFind (fun t -> t.Id = TaskId id) with

| Some task ->

if task.Status = Cancelled then

return Error (InvalidOperation "Cannot toggle cancelled task")

else

let newStatus = if task.Status = Completed then Pending else Completed

tasks <- tasks |> List.map (fun t -> if t.Id = TaskId id then { t with Status = newStatus } else t)

eventAgent.Post(TaskStatusChanged (TaskId id, newStatus))

return Ok ()

| None -> return Error (TaskNotFound id)

}

member this.CancelTask id =

async {

match tasks |> List.tryFind (fun t -> t.Id = TaskId id) with

| Some task ->

if task.Status = Cancelled then

return Error (InvalidOperation "Task already cancelled")

else

tasks <- tasks |> List.map (fun t -> if t.Id = TaskId id then { t with Status = Cancelled } else t)

eventAgent.Post(TaskCancelled (TaskId id))

return Ok ()

| None -> return Error (TaskNotFound id)

}

member this.ListTasks() =

async { return tasks }

member this.FilterByPriority priority =

async {

match priority.ToLower() with

| "low" -> return Ok (tasks |> List.filter (fun t -> t.Priority = Low))

| "medium" -> return Ok (tasks |> List.filter (fun t -> t.Priority = Medium))

| "high" -> return Ok (tasks |> List.filter (fun t -> t.Priority = High))

| \_ -> return Error (InvalidPriority $"Invalid priority: {priority}")

}

member this.FilterByStatus status =

async {

match status.ToLower() with

| "pending" -> return Ok (tasks |> List.filter (fun t -> t.Status = Pending))

| "completed" -> return Ok (tasks |> List.filter (fun t -> t.Status = Completed))

| "cancelled" -> return Ok (tasks |> List.filter (fun t -> t.Status = Cancelled))

| \_ -> return Error (InvalidOperation $"Invalid status: {status}")

}

member this.Summary() =

async {

let total = tasks |> PSeq.length

let completed = tasks |> PSeq.filter (fun t -> t.Status = Completed) |> PSeq.length

let highPriority = tasks |> PSeq.filter (fun t -> t.Priority = High) |> PSeq.length

let overdue = tasks |> PSeq.filter (fun t ->

match t.DueDate with

| Some days -> days < 0.0<day>

| None -> false) |> PSeq.length

return (total, completed, highPriority, overdue)

}

member this.SaveTasks filePath =

async {

try

let json = sprintf "[%s]" (tasks |> List.map toJson |> String.concat ",")

do! File.WriteAllTextAsync(filePath, json) |> Async.AwaitTask

return Ok ()

with

| ex -> return Error (InvalidOperation $"Error saving tasks: {ex.Message}")

}

member this.LoadTasks filePath =

async {

try

let! content = File.ReadAllTextAsync(filePath) |> Async.AwaitTask

let jsonTasks = TaskJson.Parse(content)

tasks <- jsonTasks |> Array.map (fun jt ->

{ Id = TaskId jt.Id

Description = jt.Description

Priority = match jt.Priority with

| "Low" -> Low

| "Medium" -> Medium

| \_ -> High

Status = match jt.Status with

| "Pending" -> Pending

| "Completed" -> Completed

| \_ -> Cancelled

DueDate = if jt.DueDate.IsSome then Some (jt.DueDate.Value \* 1.0<day>) else None

}) |> Array.toList

idCounter <- (tasks |> List.map (fun t -> let (TaskId id) = t.Id in id) |> List.max) + 1

return Ok ()

with

| ex -> return Error (InvalidOperation $"Error loading tasks: {ex.Message}")

}

member this.GetEvents() =

async {

let! events = eventAgent.PostAndAsyncReply(fun reply -> GetNotifications reply)

return events

}

member this.Tasks = tasks // For testing

**Changes from Chapter 10**

* **Interface Updates**:
  + ListTasks, FilterByPriority, and FilterByStatus return Async<Task list> or Async<Result<Task list, DomainError>> for API compatibility.
  + Summary returns Async<(int \* int \* int \* int)> for structured data.
* **Simplified Output**: Removed console printing from ListTasks, FilterByPriority, and FilterByStatus, as the API will handle display.
* **Preserved Features**: DDD (TaskId, TaskStatus), ROP (DomainError), async operations, and event logging.

## Web API: Giraffe Backend

The web project hosts the API and serves the React frontend.

**Setup Web Project**

cd TaskManagerFSharp

mkdir TaskManagerWeb

cd TaskManagerWeb

dotnet new web -lang "F#"

dotnet add reference ../TaskManagerFSharp.fsproj

dotnet add package Giraffe

**Program.fs (TaskManagerWeb)**

open Microsoft.AspNetCore.Builder

open Microsoft.AspNetCore.Http

open Microsoft.Extensions.DependencyInjection

open Giraffe

open System.Text.Json

open TaskManager

type TaskInput = {

Description: string

Priority: string

DueDate: string

}

let apiRoutes (manager: ITaskManager) : HttpHandler =

let jsonOptions = JsonSerializerOptions(JsonSerializerDefaults.Web)

let handleResult next ctx result =

match result with

| Ok data -> successful OK data next ctx

| Error err ->

let errorMsg = match err with

| InvalidDescription msg -> msg

| InvalidPriority msg -> msg

| InvalidDueDate msg -> msg

| TaskNotFound id -> sprintf "Task %d not found" id

| InvalidOperation msg -> msg

request (fun \_ -> setStatusCode 400 >=> text errorMsg) next ctx

choose [

GET >=> route "/api/tasks" >=> fun next ctx ->

task {

let! tasks = manager.ListTasks()

return! json tasks jsonOptions next ctx

}

POST >=> route "/api/tasks" >=> fun next ctx ->

task {

let! input = ctx.BindJsonAsync<TaskInput>()

let! result = manager.AddTask input.Description input.Priority input.DueDate

return! handleResult next ctx result

}

PUT >=> routef "/api/tasks/%i/toggle" (fun id -> fun next ctx ->

task {

let! result = manager.ToggleComplete id

return! handleResult next ctx result

})

PUT >=> routef "/api/tasks/%i/cancel" (fun id -> fun next ctx ->

task {

let! result = manager.CancelTask id

return! handleResult next ctx result

})

GET >=> routef "/api/tasks/priority/%s" (fun priority -> fun next ctx ->

task {

let! result = manager.FilterByPriority priority

return! handleResult next ctx (result |> Result.map (fun tasks -> json tasks jsonOptions))

})

GET >=> routef "/api/tasks/status/%s" (fun status -> fun next ctx ->

task {

let! result = manager.FilterByStatus status

return! handleResult next ctx (result |> Result.map (fun tasks -> json tasks jsonOptions))

})

GET >=> route "/api/summary" >=> fun next ctx ->

task {

let! (total, completed, highPriority, overdue) = manager.Summary()

let summary = {| Total = total; Completed = completed; HighPriority = highPriority; Overdue = overdue |}

return! json summary jsonOptions next ctx

}

GET >=> route "/api/events" >=> fun next ctx ->

task {

let! events = manager.GetEvents()

return! json events jsonOptions next ctx

}

route "/" >=> htmlFile "wwwroot/index.html"

]

let configureApp (app: IApplicationBuilder) =

app.UseStaticFiles()

.UseGiraffe(apiRoutes (TaskManager() :> ITaskManager))

let configureServices (services: IServiceCollection) =

services.AddGiraffe() |> ignore

services.AddCors(fun options ->

options.AddPolicy("AllowAll", fun builder ->

builder.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader() |> ignore)) |> ignore

[<EntryPoint>]

let main argv =

let builder = WebApplication.CreateBuilder(argv)

configureServices builder.Services

let app = builder.Build()

app.UseCors("AllowAll") |> ignore

configureApp app

app.Run()

0

**Key Backend Features**

* **API Endpoints**:
  + GET /api/tasks: Lists all tasks.
  + POST /api/tasks: Adds a task (expects JSON { "description": "", "priority": "", "dueDate": "" }).
  + PUT /api/tasks/{id}/toggle: Toggles task status.
  + PUT /api/tasks/{id}/cancel: Cancels a task.
  + GET /api/tasks/priority/{priority}: Filters by priority.
  + GET /api/tasks/status/{status}: Filters by status.
  + GET /api/summary: Returns task summary.
  + GET /api/events: Returns event log.
* **Error Handling**:
  + handleResult: Converts Result to HTTP responses (200 for Ok, 400 for Error).
  + Uses ROP from Chapter 10 to manage domain errors.
* **CORS**: Enabled with AllowAll policy for frontend development (restrict in production).
* **JSON Serialization**: Uses System.Text.Json with web defaults (camelCase).
* **Static Files**: Serves index.html for the React frontend.

**Frontend: React with Tailwind CSS**

The frontend is a single-page React app, served from wwwroot/index.html, using CDNs for React and Tailwind CSS.

**Create wwwroot/index.html**

In TaskManagerWeb/wwwroot, create index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Task Manager</title>

<script src="https://cdn.jsdelivr.net/npm/react@18.2.0/umd/react.development.js"></script>

<script src="https://cdn.jsdelivr.net/npm/react-dom@18.2.0/umd/react-dom.development.js"></script>

<script src="https://cdn.tailwindcss.com"></script>

<script src="https://cdn.jsdelivr.net/npm/axios@1.6.7/dist/axios.min.js"></script>

<script src="https://unpkg.com/@babel/standalone@7.23.9/babel.min.js"></script>

</head>

<body class="bg-gray-100">

<div id="root" class="container mx-auto p-4"></div>

<script type="text/babel">

const { useState, useEffect } = React;

function TaskManagerApp() {

const [tasks, setTasks] = useState([]);

const [description, setDescription] = useState('');

const [priority, setPriority] = useState('Medium');

const [dueDate, setDueDate] = useState('');

const [filterPriority, setFilterPriority] = useState('');

const [filterStatus, setFilterStatus] = useState('');

const [summary, setSummary] = useState(null);

const [events, setEvents] = useState([]);

const fetchTasks = async () => {

const response = await axios.get('/api/tasks');

setTasks(response.data);

};

const fetchSummary = async () => {

const response = await axios.get('/api/summary');

setSummary(response.data);

};

const fetchEvents = async () => {

const response = await axios.get('/api/events');

setEvents(response.data);

};

useEffect(() => {

fetchTasks();

fetchSummary();

fetchEvents();

}, []);

const addTask = async (e) => {

e.preventDefault();

try {

await axios.post('/api/tasks', { description, priority, dueDate });

setDescription('');

setDueDate('');

fetchTasks();

fetchEvents();

} catch (error) {

alert(error.response?.data || 'Error adding task');

}

};

const toggleTask = async (id) => {

try {

await axios.put(`/api/tasks/${id}/toggle`);

fetchTasks();

fetchEvents();

} catch (error) {

alert(error.response?.data || 'Error toggling task');

}

};

const cancelTask = async (id) => {

try {

await axios.put(`/api/tasks/${id}/cancel`);

fetchTasks();

fetchEvents();

} catch (error) {

alert(error.response?.data || 'Error cancelling task');

}

};

const filterByPriority = async () => {

if (!filterPriority) {

fetchTasks();

return;

}

try {

const response = await axios.get(`/api/tasks/priority/${filterPriority}`);

setTasks(response.data);

} catch (error) {

alert(error.response?.data || 'Error filtering by priority');

}

};

const filterByStatus = async () => {

if (!filterStatus) {

fetchTasks();

return;

}

try {

const response = await axios.get(`/api/tasks/status/${filterStatus}`);

setTasks(response.data);

} catch (error) {

alert(error.response?.data || 'Error filtering by status');

}

};

return (

<div className="max-w-4xl mx-auto">

<h1 className="text-3xl font-bold mb-4 text-center">Task Manager</h1>

{/\* Add Task Form \*/}

<div className="bg-white p-6 rounded-lg shadow-md mb-6">

<h2 className="text-xl font-semibold mb-4">Add Task</h2>

<form onSubmit={addTask} className="space-y-4">

<div>

<label className="block text-sm font-medium">Description</label>

<input

type="text"

value={description}

onChange={(e) => setDescription(e.target.value)}

className="w-full p-2 border rounded"

required

/>

</div>

<div>

<label className="block text-sm font-medium">Priority</label>

<select

value={priority}

onChange={(e) => setPriority(e.target.value)}

className="w-full p-2 border rounded"

>

<option>Low</option>

<option>Medium</option>

<option>High</option>

</select>

</div>

<div>

<label className="block text-sm font-medium">Due Date (days)</label>

<input

type="text"

value={dueDate}

onChange={(e) => setDueDate(e.target.value)}

className="w-full p-2 border rounded"

placeholder="Enter number of days or leave empty"

/>

</div>

<button type="submit" className="bg-blue-500 text-white p-2 rounded hover:bg-blue-600">

Add Task

</button>

</form>

</div>

{/\* Filters \*/}

<div className="bg-white p-6 rounded-lg shadow-md mb-6">

<h2 className="text-xl font-semibold mb-4">Filters</h2>

<div className="flex space-x-4">

<div>

<label className="block text-sm font-medium">Priority Filter</label>

<select

value={filterPriority}

onChange={(e) => { setFilterPriority(e.target.value); filterByPriority(); }}

className="p-2 border rounded"

>

<option value="">All</option>

<option>Low</option>

<option>Medium</option>

<option>High</option>

</select>

</div>

<div>

<label className="block text-sm font-medium">Status Filter</label>

<select

value={filterStatus}

onChange={(e) => { setFilterStatus(e.target.value); filterByStatus(); }}

className="p-2 border rounded"

>

<option value="">All</option>

<option>Pending</option>

<option>Completed</option>

<option>Cancelled</option>

</select>

</div>

</div>

</div>

{/\* Task List \*/}

<div className="bg-white p-6 rounded-lg shadow-md mb-6">

<h2 className="text-xl font-semibold mb-4">Tasks</h2>

{tasks.length === 0 ? (

<p>No tasks available.</p>

) : (

<ul className="space-y-2">

{tasks.map(task => (

<li key={task.id} className="flex justify-between items-center p-2 border-b">

<div>

<span className="font-medium">{task.description}</span>

<span className="text-gray-600"> [{task.priority}] - {task.status}</span>

<span className="text-gray-500">

{task.dueDate ? ` (Due in ${task.dueDate} days)` : ' (No due date)'}

</span>

</div>

<div>

<button

onClick={() => toggleTask(task.id)}

className="bg-green-500 text-white px-3 py-1 rounded mr-2 hover:bg-green-600"

>

Toggle

</button>

<button

onClick={() => cancelTask(task.id)}

className="bg-red-500 text-white px-3 py-1 rounded hover:bg-red-600"

>

Cancel

</button>

</div>

</li>

))}

</ul>

)}

</div>

{/\* Summary \*/}

{summary && (

<div className="bg-white p-6 rounded-lg shadow-md mb-6">

<h2 className="text-xl font-semibold mb-4">Summary</h2>

<p>Total Tasks: {summary.total}</p>

<p>Completed: {summary.completed}</p>

<p>High Priority: {summary.highPriority}</p>

<p>Overdue: {summary.overdue}</p>

</div>

)}

{/\* Events \*/}

<div className="bg-white p-6 rounded-lg shadow-md">

<h2 className="text-xl font-semibold mb-4">Event Log</h2>

{events.length === 0 ? (

<p>No events.</p>

) : (

<ul className="space-y-2">

{events.map((event, index) => (

<li key={index} className="text-gray-700">{event}</li>

))}

</ul>

)}

</div>

</div>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<TaskManagerApp />);

</script>

</body>

</html>

**Frontend Features**

* **Components**:
  + Form to add tasks with description, priority, and due date.
  + Dropdowns for filtering by priority and status.
  + Task list with toggle and cancel buttons.
  + Summary and event log displays.
* **State Management**:
  + Uses useState for tasks, inputs, filters, summary, and events.
  + useEffect fetches initial data (tasks, summary, events).
* **API Integration**:
  + Uses axios for HTTP requests to the Giraffe API.
  + Handles errors via alerts (simplified for demo).
* **Styling**:
  + Tailwind CSS for responsive, modern UI.
  + Card-based layout with shadows and spacing.
* **Reactivity**:
  + Updates UI on task changes (add, toggle, cancel, filter).

**Update Tests.fs**

Update tests to reflect the API-friendly interface:

open Xunit

open System

open TaskManager

type TaskManagerTests() =

let manager = TaskManager() :> ITaskManager

[<Fact>]

let ``AddTask adds valid task`` () =

async {

let! result = manager.AddTask "Test Task" "High" "5"

match result with

| Ok () ->

let! tasks = manager.ListTasks()

let task = tasks |> List.find (fun t -> t.Description = "Test Task")

Assert.Equal("Test Task", task.Description)

Assert.Equal(High, task.Priority)

Assert.Equal(Some 5.0<day>, task.DueDate)

Assert.Equal(Pending, task.Status)

| Error e -> Assert.Fail($"Expected Ok, got {e}")

} |> Async.RunSynchronously

[<Fact>]

let ``AddTask fails on empty description`` () =

async {

let! result = manager.AddTask "" "Medium" ""

match result with

| Error (InvalidDescription \_) -> ()

| \_ -> Assert.Fail("Expected InvalidDescription error")

} |> Async.RunSynchronously

[<Fact>]

let ``ToggleComplete updates status`` () =

async {

do! manager.AddTask "Toggle Test" "Medium" "" |> Async.Ignore

let id = (manager.Tasks |> List.head).Id |> fun (TaskId id) -> id

let! result = manager.ToggleComplete id

match result with

| Ok () ->

let! tasks = manager.ListTasks()

let task = tasks |> List.find (fun t -> t.Id = TaskId id)

Assert.Equal(Completed, task.Status)

| Error e -> Assert.Fail($"Expected Ok, got {e}")

} |> Async.RunSynchronously

[<Fact>]

let ``CancelTask updates status`` () =

async {

do! manager.AddTask "Cancel Test" "Low" "" |> Async.Ignore

let id = (manager.Tasks |> List.head).Id |> fun (TaskId id) -> id

let! result = manager.CancelTask id

match result with

| Ok () ->

let! tasks = manager.ListTasks()

let task = tasks |> List.find (fun t -> t.Id = TaskId id)

Assert.Equal(Cancelled, task.Status)

| Error e -> Assert.Fail($"Expected Ok, got {e}")

} |> Async.RunSynchronously

[<Fact>]

let ``Events record task actions`` () =

async {

do! manager.AddTask "Event Test" "High" "" |> Async.Ignore

let id = (manager.Tasks |> List.head).Id |> fun (TaskId id) -> id

do! manager.ToggleComplete id |> Async.Ignore

let! events = manager.GetEvents()

Assert.Contains("Created task", events |> String.concat " ")

Assert.Contains("Changed task", events |> String.concat " ")

} |> Async.RunSynchronously

**Deployment Considerations**

* **Hosting**: Deploy to Azure, AWS, or a Linux server with dotnet publish.
* **CORS**: Restrict AllowAll policy in production (e.g., allow specific origins).
* **Storage**: Replace JSON file with a database (e.g., SQLite with SqlClient type provider).
* **Security**: Add authentication (e.g., JWT with ASP.NET Core Identity).
* **Performance**: Optimize PSeq usage for large datasets; consider caching.

**Running the Application**

1. **Console App** (optional, unchanged from Chapter 10):

cd TaskManagerFSharp

dotnet run

1. **Web App**:

cd TaskManagerWeb

dotnet run

* Access http://localhost:5000 to view the React frontend.
* Test API with Postman or curl, e.g.:

curl -X POST http://localhost:5000/api/tasks -H "Content-Type: application/json" -d '{"description":"Write report","priority":"High","dueDate":"3"}'

1. **Tests**:

cd Tests

dotnet test

**Example Interaction**

* Open http://localhost:5000.
* Add a task (e.g., "Write report", High, 3 days).
* Toggle or cancel tasks via buttons.
* Filter by priority (e.g., High) or status (e.g., Pending).
* View summary and event log.
* **Sample API Response** (GET /api/tasks):

[

{"id":1,"description":"Write report","priority":"High","status":"Pending","dueDate":3}

]

**In-Depth Explanation**

* **DDD**:
  + TaskId, TaskStatus, and Task model the domain precisely.
  + TaskEvent captures changes for auditing.
* **ROP**:
  + DomainError ensures explicit error handling.
  + API converts errors to HTTP 400 responses.
* **Async**:
  + All operations are asynchronous, using Async and task {} for Giraffe compatibility.
* **MailboxProcessor**:
  + Logs events, accessible via /api/events.
* **Giraffe**:
  + Functional routing with choose and routef.
  + Integrates seamlessly with ITaskManager.
* **React Frontend**:
  + Reusable components for form, filters, list, summary, and events.
  + Tailwind CSS provides responsive, modern styling.
  + Axios handles API calls with error feedback.
* **Testing**:
  + Verifies API endpoints and domain logic.
  + Ensures error cases (e.g., invalid inputs) are handled.

**Why This Approach?**

* **F# Strengths**:
  + DDD and ROP ensure a robust, type-safe domain model.
  + Async and PSeq provide scalability.
  + Giraffe’s functional style aligns with F#’s paradigms.
* **React Integration**:
  + SPA architecture simplifies client-side logic.
  + Tailwind CSS enables rapid UI development.
* **Extensibility**:
  + Easy to add authentication, database storage, or additional endpoints.
* **Maintainability**:
  + Clear separation of concerns (domain, API, UI).
  + Tests ensure reliability.

## Experiment in FSI

Test API logic:

let manager = TaskManager() :> ITaskManager

Async.RunSynchronously (manager.AddTask "Test" "High" "5") // Ok ()

Async.RunSynchronously (manager.ListTasks()) // [{ Id = TaskId 1; Description = "Test"; ... }]

**Future Enhancements**

* **Database**: Use SqlClient type provider for persistence.
* **Authentication**: Integrate JWT or OAuth.
* **Real-Time Updates**: Add WebSockets for live task updates.
* **Frontend Build**: Use Vite or Webpack for production builds.

This chapter completes the task manager, showcasing F#’s power in full-stack development. You’ve built a robust, scalable web app, ready for real-world use or further enhancement.