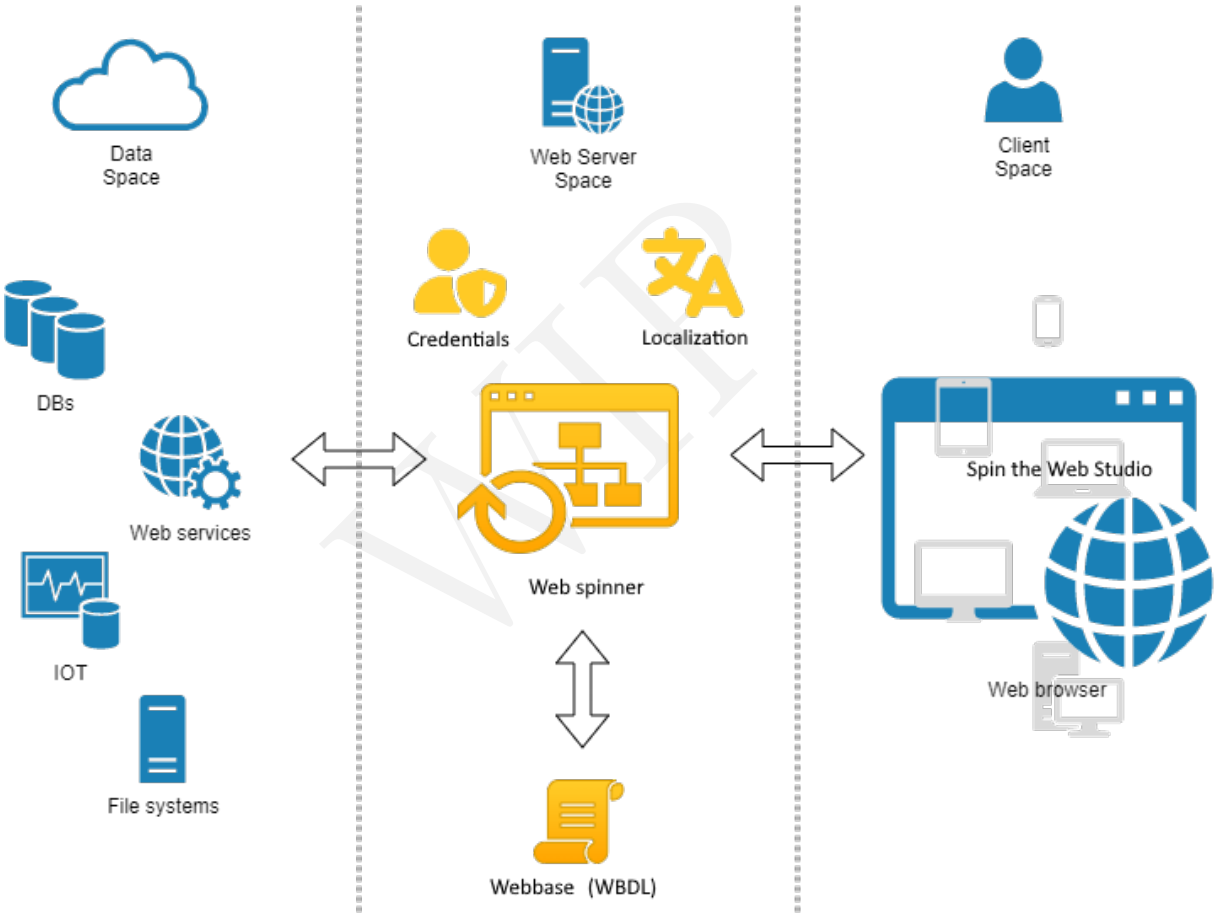
Spin the Web

Weaving Digital Portals



Spin the Web Foundation

Visualize, then Realize: Weaving Web Portals to Shape Your Brand's Identity.

License

You are free to share and adapt the material for any purpose, even commercially, provided that you give appropriate credit and distribute your contributions under the same license as the original. For attribution, cite the work as: "Spin the Web (Morphic Edition), Giancarlo Trevisan, 2025."

Note: This license applies to the book's content. The Spin the Web framework itself is intended to remain free to use under the stewardship of the Spin the Web Foundation, which publishes and maintains the specifications and reference implementations.

To the Internet, a neverending source of knowledge.
To the open source community, a wonderful group that shares its thoughts in hacks and code.

Abstract

Spin the Web is a framework for building enterprise web portals ("portals") that virtualize the company brand—*eBranding*. It addresses the persistent challenge of unifying heterogeneous enterprise systems (ERP, CRM, BPMS, and MRP systems) behind a single, role-aware digital channel, providing consistent abstractions over disparate backends.

The framework comprises three core components:

1. **Webbase Description Language (WBDL)**: A declarative language for modeling portal structure, content, and behavior.
2. **Web Spinner**: An engine that interprets webbases (modular portal definitions written in WBDL) to dynamically generate personalized user experiences with real-time content delivery and role-based authorization.
3. **Spin the Web Studio**: An authoring webbaselet for in-place editing of webbases; it also serves as a laboratory for exercising the Web Spinner.

The project also introduces:

- **Webbaselets**: Modular, self-contained WBDL fragments that encapsulate integrations with enterprise systems.
- **Webbase Placeholders Language (WBPL)**: A security-conscious templating mechanism for parameterized queries.
- **Webbase Layout Language (WBLL)**: A token-based templating language for presentation and rendering.
- **Virtualized Company paradigm**: A unified portal interface for customers, employees, suppliers, partners, and governance stakeholders, with the primary objective of digitally harboring the brand (*eBranding*).

This book blends foundations with practical guidance for developers, integrators, and technology leaders modernizing digital channels.

Keywords: enterprise portals; WBDL; webbaselets; WBPL; WBLL; eBranding; STW; Foundation.

Preface

Why This Book

In the rapidly evolving landscape of enterprise software, organizations find themselves trapped in a web of disparate systems, each with its own interface, data model, and user experience paradigm. Employees struggle to navigate between multiple applications, customers face fragmented touchpoints, and partners encounter inconsistent integration patterns. The promise of digital transformation often falls short because these systems remain fundamentally siloed.

Spin the Web emerged from real-world frustration with this fragmentation. After building custom integrations, developing multiple user interfaces, and watching organizations struggle with the complexity of their own digital ecosystems, it became clear that a fundamentally different approach was needed.

What Makes This Different

This book introduces a paradigm shift: instead of trying to integrate disparate systems at the data level, let's integrate them at the experience level! The `WBDL` specification provides a way to describe portal structures that can encompass any type of enterprise system. The *Web Spinner* engine interprets WBDL to enable real-time content delivery. The *Spin the Web Studio* provides the tools to build and maintain these portals efficiently.

What sets this approach apart is the concept of the "virtualized company"—a single, coherent digital interface that adapts to each user's role and needs, whether they are a customer, employee, supplier, or partner. This isn't just another portal framework; it's a complete rethinking of how organizations should present themselves digitally, i.e., how they should be *e-branded*.

Who Should Read This Book

This book is written for professional software developers, enterprise architects, and technology leaders who are responsible for building or maintaining complex web-based systems. While the concepts are accessible to developers with basic web development experience, the focus is on enterprise-grade solutions that require sophisticated understanding of system integration, security, and scalability.

Specifically, this book will be valuable to:

- Full-stack developers building enterprise web applications
- System architects designing portal solutions

- Development team leads planning integration strategies
- Technology consultants working with enterprise clients
- CIOs and CTOs evaluating portal technologies

How This Book Is Organized

The book follows a logical progression from concepts to implementation:

Part I establish the theoretical foundations, explaining the problem space and introducing the core concepts of Spin the Web.

Part II dive deep into the core trio of the Blueprint, the Engine, and the Workshop: the `WBDL` language specification, the *Web Spinner* engine architecture, and the *Spin the Web Studio* development environment.

Part III focuses on web portal development—design, structure, user experience, and best practices. Where appropriate, the framework's features and approaches are illustrated in context.

Part IV documents future directions, this part of the book will continue to evolve.

Each part builds upon previous concepts while remaining sufficiently self-contained for reference use.

About the Spin the Web Foundation

The mission of the Spin the Web Foundation is to divulge, manage, and evolve the Spin the Web framework while keeping it free to use. The Foundation stewards specifications (`WBDL`, `WBPL`, `WBLL`), reference implementations (the *Web Spinner* and *Spin the Web Studio*), and ecosystem guidance so that individuals and organizations can adopt, extend, and interoperate without vendor lock-in.

To that end, this book documents the conceptual model and provides practical, implementation-oriented guidance. Where contributions, governance, or trademark policies are relevant, they are handled transparently by the Foundation.

Project repository: `https://github.com/spintheweb`.

Acknowledgments

The practical insights in this book were refined through collaboration with enterprises, integration partners, and the broader community of developers working to solve real-world portal challenges.

Giancarlo Trevisan

September 14, 2025

Contents

Part I

The Philosophy

Introduction to Part I

> *"The best way to predict the future is to create it."*
> — Peter Drucker

In this opening part, we explore the historical origins and fundamental challenges that led to Spin the Web, and introduce the conceptual foundations that guide its approach to enterprise portal development.

We begin with the project's genesis in the late 1990s, tracing its evolution from a practical business challenge to the framework. We then examine the current landscape of enterprise software, where organizations struggle with fragmented user experiences across multiple systems. Next, we introduce the concept of the "virtualized company"—a unified digital interface that adapts to each stakeholder's role and needs.

Finally, we provide an overview of the core architecture that makes this vision possible: the `WBDL` specification for describing portal structures, the *Web Spinner* engine for dynamic content delivery, and the *Spin the Web Studio* for development and maintenance.

Chapter 1: Genesis and Historical Context (chapter 1) – Explores the real-world origins of Spin the Web, from its birth in an Italian jewelry business in the late 1990s through the evolution of the eBranding concept. This chapter provides crucial context for understanding both the technical innovations and business philosophy underlying the framework.

Chapter 2: Introduction to Enterprise Portal Challenges (chapter 2) – Examines the contemporary challenges facing modern enterprises in their digital transformation journeys and introduces the foundational concepts of Spin the Web.

Chapter 3: Web Portals as Virtualized Companies (chapter 3) – Introduces the concept of web portals as "virtualized companies"—unified digital interfaces that provide role-based access to all organizational functions, serving diverse stakeholders from employees to customers to partners.

Chapter 4: The Blueprint, Engine, and Workshop (chapter 4) – Provides an overview of the core architecture that makes the virtualized company vision possible: the `WBDL` specification (Blueprint), the *Web Spinner* engine (Engine), and the *Spin the Web Studio* (Workshop).

These foundational concepts will prepare you for the detailed technical exploration that follows in subsequent parts of the book.

Mastering these specifications is the first step toward building robust, enterprise-grade portals. This section serves as the definitive reference for the framework's "Blueprint," "Engine," and "Workshop."

Chapter 1

Genesis and HHistory

This chapter explores the real-world origins of Spin the Web, tracing its evolution from a practical business challenge in the late 1990s to the systematic framework presented in this book. Understanding this genesis provides crucial context for appreciating both the technical innovations and the business philosophy that underpin the project.

1.1 The Italian Jewelry Business Challenge

In the late 1990s, the digital landscape was vastly different from today's interconnected world. Enterprise software was fragmented, web technologies were in their infancy, and businesses struggled to integrate their complex operational structures into cohesive digital experiences.

It was during this period that the seeds of Spin the Web were planted through a consulting engagement with an Italian jewelry business. This company represented the complexity typical of many enterprises: a nationwide sales force, distributed points of sale, international suppliers, in-house and national jewelry designers and manufacturers, a help desk, and a call center. Each component of this business ecosystem had its own requirements, processes, and stakeholders.

The challenge was clear: how to network this intricate structure in a way that would enable seamless collaboration, efficient information flow, and unified business processes across all participants in the ecosystem.

1.2 From Lotus Notes to Web Technologies

1.2.1 The Lotus Notes Solution

To address their networking needs, the company's IT department had implemented **Lotus Notes**, a collaborative client-server software platform developed at IBM. This choice was both sensible and forward-thinking for its time:

- **Effective Use of Available Connectivity**: Lotus Notes made intelligent use of the limited connectivity options available in the late 1990s
- **Data Replication**: The platform successfully replicated data stored in a proprietary NoSQL database across distributed locations

- **Development Environment**: It provided a respectable development environment for building front-ends to interact with business data
- **Collaborative Features**: Beyond data management, Notes offered collaborative features that enhanced team communication
- **Multi-Purpose Platform**: The sales force and points of sale used it to browse product catalogs and place orders, while the help desk and call center leveraged it as a knowledge base

1.2.2 The Hybrid Solution Challenge

When tasked with developing a solution to interface with the manufacturers, a significant technical challenge emerged. Lotus Notes did not handle SQL data particularly well, creating a mismatch between the platform's strengths and the manufacturers' data systems.

The solution adopted was a hybrid approach—a compromise that bridged the gap between the Notes environment and SQL-based manufacturing systems. While this approach was functional and met the immediate business needs, it highlighted a fundamental limitation: the difficulty of creating truly unified interfaces when working with disparate systems and technologies.

This experience planted the first seeds of what would later become the *webbaselet* concept—the idea that different business systems could be unified through a common interface layer without requiring them to abandon their underlying architectures.

1.3 The Dynamic Web Pages Revelation

1.3.1 The Internet Evolution

As this project unfolded, the Internet was undergoing a revolutionary transformation. Dynamic web pages were making their debut[1], fundamentally changing how web applications could be conceived and implemented.

A dynamic web page represented a paradigm shift: rather than serving static HTML content, web servers could now assemble pages on-demand in response to specific client requests. This concept proved to be profoundly powerful and opened up entirely new possibilities for enterprise applications.

1.3.2 The Conceptual Breakthrough

The revelation came through understanding the full implications of dynamic page generation:

- **Universal Data Access**: Data sources in general could be queried by the web server in response to client requests
- **On-Demand Rendering**: Fetched data could be rendered as HTML before being sent to the client
- **Intuitive Data Interaction**: Clients could inspect data intuitively and perform insertions, updates, and deletions (CRUD operations)

[1]The technologies primarily referenced are ASP and PHP, which made dynamic pages easier to build. CGI had been available for some time and could accomplish similar functionality, but these newer technologies significantly lowered the barrier to entry.

- **Unified Interface**: The same web page could host data coming from disparate data sources in a coherent, tailored graphical user interface

This led to a pivotal insight: web technologies could be used not just for public-facing websites, but as the foundation for comprehensive enterprise portals.

1.4 The Portal Vision Emerges

1.4.1 Beyond Traditional Websites

While developing a proof of concept for this dynamic approach, a transformative idea crystallized: use web technologies inside the company to build a web site—later termed a **portal**—whose target audience extended far beyond the general public.

This portal would serve:

- **Company Employees**: Access to internal systems, processes, and information
- **Sales Force**: Product catalogs, order management, and customer relationship tools
- **Suppliers**: Integration points for inventory, orders, and collaboration
- **Customers**: Self-service capabilities and direct business interaction

The vision was compelling: a single, web-based interface that could accommodate the diverse needs of all stakeholders in the business ecosystem, while maintaining security, personalization, and role-based access control.

1.4.2 The Systematic Approach Imperative

The idea was undoubtedly sound, but implementing it successfully required more than ad-hoc development. What was needed was a **systematic approach**—a comprehensive framework that could handle the complexity of enterprise portals in a consistent, scalable, and maintainable way.

This realization marked the beginning of what would eventually become Spin the Web's core mission: developing the tools, languages, and methodologies necessary to transform the portal vision into practical reality.

1.5 The Birth of WBDL

1.5.1 Language Requirements

The systematic approach demanded the definition of a specialized language capable of describing entire web sites with unprecedented detail and flexibility. This language would need to handle:

- **Complex Site Structure**: Hierarchical organization of areas, pages, and content
- **Routing Logic**: URL-to-content mapping and navigation flows
- **Authorization Rules**: Role-based access control and visibility management
- **Internationalization**: Multi-language support for global enterprises
- **Data Integration**: Connections to diverse data sources and systems

This language would need to be structured enough to handle complex enterprise realities while remaining flexible enough to evolve with changing business needs. This language was named **WBDL** (Webbase Description Language).

1.5.2 The Interpreter Requirement

Alongside the language definition, a second critical requirement emerged: the development of an **interpreter**—the Web Spinner—that could take a URL request, fetch the associated `WBDL` fragment, and render it dynamically.

The analogy was clear and powerful: just as HTML is interpreted by a web browser to create user-facing web pages, `WBDL` would be interpreted by a Web Spinner to create complete portal experiences.

This interpreter would need to:

- Parse and understand `WBDL` documents
- Handle user authentication and authorization
- Manage data source connections and queries
- Render content appropriately for different users and contexts
- Maintain high performance under enterprise-scale loads

1.6 Evolution and Persistence of the Vision

1.6.1 Timeless Principles

Since its inception, `WBDL` has demonstrated remarkable resilience and relevance. The core principles identified in the late 1990s have proven to be enduring:

- **Descriptive Power**: `WBDL` can describe web sites with the same ease today as it could in the past
- **Technology Independence**: The framework remains relevant despite the rapidly evolving Internet landscape
- **Systematic Consistency**: Much like HTML, `WBDL` provides a stable foundation that transcends specific technological implementations

This persistence suggests that the project identified fundamental patterns and requirements that are intrinsic to enterprise portal development, rather than merely addressing temporary technological limitations.

1.6.2 Continuous Refinement

While the core vision has remained stable, Spin the Web has been able to manage new insights, technologies, and best practices. This evolution has been guided by real-world implementation experiences and the changing needs of enterprise software development.

The framework's ability to adapt while maintaining its essential character speaks to the soundness of its foundational principles and architectural decisions.

1.7 The eBranding Concept

1.7.1 Defining eBranding

The practical experiences and technical innovations of Spin the Web eventually crystallized into a broader business philosophy: **eBranding**.

eBranding is defined as *the act of virtualizing all virtualizable aspects of an entity, be it an organization, a company, a trade, or a group.* This concept extends far beyond traditional web presence or digital marketing.

1.7.2 The Ultimate eBranding Goal

The ultimate goal of eBranding is to build a **web portal**—a "website on steroids"—whose target audience encompasses:

- **Customers**: Primary market and service recipients
- **Suppliers**: Business partners and service providers
- **Shareholders/Investors**: Financial stakeholders and governance bodies
- **Regulators**: Compliance and oversight entities
- **Partners**: Strategic allies and collaborators
- **Distributors**: Channel partners and intermediaries
- **Contractors**: Service providers and consultants
- **Employees**: Internal workforce and management
- **Community**: Local and industry communities
- **Other Web Portals**: System-to-system integration points

This portal serves as an **all-encompassing channel for any kind of interaction with the entity**—a digital manifestation of the organization that evolves continuously with the business itself.

1.7.3 Integration Philosophy

Spin the Web's approach to eBranding is fundamentally integrative rather than disruptive. The framework's intentions are:

- **Not to Replace**: Existing systems and processes remain valuable and should be preserved where appropriate
- **But to Integrate**: Everything should be brought together in a unified environment
- **Enable Evolution**: The brand and its digital presence should be able to evolve naturally over time
- **Leverage the Internet**: Use Internet technologies as the foundation for this integration

This philosophy recognizes that most enterprises have significant investments in existing systems and processes. Rather than requiring wholesale replacement, Spin the Web provides a framework for unifying these disparate elements into a coherent whole.

1.8 The Evolution of Webbase Concept

1.8.1 From Relational Database to Schema-Defined Structures

The conceptual foundation of Spin the Web has its roots in a pioneering approach first developed in 1998. The initial breakthrough came with the introduction of the term **webbase**—a specialized relational database whose schema was specifically designed to define and manage web structures.

This original webbase encompassed all the essential aspects of web presence:

Structure : Hierarchical organization of areas, pages, and content
Content : Text, media, and interactive elements
Layout : Visual presentation and responsive design patterns
Localization : Multi-language support and cultural adaptation
Navigation : User journey flows and interaction patterns
Security : Access control and authorization frameworks

1.8.2 The Schema Revolution: XML and JSON

While the relational database approach proved effective, it presented challenges for portability and interoperability. To address these limitations, the webbase concept underwent a fundamental transformation—it was formalized into a schema-based language structure.

Initially, this took the form of an XML-based language, which introduced two crucial concepts:

Webbase Description Language (WBDL) : An XML schema that formally defined how web portals should be structured, replacing the relational database schema with a portable, standardized format.

Webbaselet : A modular fragment of a webbase that could be independently developed, maintained, and integrated.

As web technologies evolved, the project embraced **JSON Schema** as a more contemporary and lightweight alternative. This migration aligned `WBDL` with modern development ecosystems, particularly those centered around JavaScript and TypeScript, while preserving the core principles of declarative structure and modularity. This transition from database schema to XML and finally to JSON Schema laid the foundation for the modular, collaborative approach that characterizes Spin the Web today.

1.8.3 Content Management System Integration

The formalization of `WBDL` positioned it as a fundamental language for Content Management Systems (CMS). This integration capability addresses several critical enterprise needs:

- **Separation of Concerns**: Content structure is independent of presentation technology
- **Version Control**: Portal configurations can be managed with standard software development practices
- **Workflow Integration**: Content approval and publishing processes integrate naturally with `WBDL` structures

- **Multi-Channel Publishing**: The same webbase can serve web, mobile, and API clients

1.9 From Vision to Reality

The journey from the late 1990s Italian jewelry business challenge to the comprehensive framework presented in this book represents more than two decades of refinement, implementation, and evolution. The core insights discovered during that initial project have proven to be both durable and expandable.

What began as a practical solution to a specific business networking challenge has evolved into a systematic approach for enterprise portal development that addresses fundamental patterns in how organizations interact with their diverse stakeholders.

The following chapters in this book detail the technical implementation of these insights, providing the practical tools and methodologies needed to transform the eBranding vision into working enterprise portals.

Next: In chapter 2, we explore the contemporary enterprise portal challenges that Spin the Web addresses, setting the stage for understanding how this historical foundation applies to today's digital landscape.

Chapter 2

Introduction to Enterprise Portal Challenges

"The single biggest problem in communication is the illusion that it has taken place."
— George Bernard Shaw

2.1 The Enterprise Software Dilemma

In today's digital economy, businesses operate through a complex web of disparate software systems—ERPs, CRMs, BPMSs, MRPs, and much more. While individually capable, these systems often create siloed user experiences, forcing employees, customers, and partners to navigate multiple interfaces to perform their tasks. This fragmentation leads to inefficiency, poor user adoption, and a disjointed view of the enterprise.

The **Spin the Web** addresses this challenge head-on. The word "Spin" is used in the sense of "to weave," as a spider dynamically weaves a web. The project is designed to weave together disparate systems and user experiences into a single, coherent whole, based on Internet technologies. It introduces a new paradigm for developing web portals centered on three core components: a specialized language, the **Webbase Description Language (WBDL)**; a server-side rendering engine, the **Web Spinner**; and a specialized *webbaselet* for editing *webbase*s, the **Spin the Web Studio**. The project's core mission is to enable the creation of unified, role-based web portals that act as a "virtualized company"—a single, coherent digital channel for all business interactions.

This document serves as the foundational guide for Spin the Web. It outlines the vision, defines the core components of the WBDL specification, and provides concrete examples of how this technology can be used to build the next generation of integrated web portals.

It is important to note that Spin the Web is a professional framework intended for full-stack developers. It is not a low-code/no-code platform for the general public, but rather a comprehensive toolset for engineers building bespoke, enterprise-grade web portals.

2.2 Understanding Web Portals

The development of a company web portal has two main goals: to **document how things are done** and to **provide a means for doing them**. It is an all-encompassing digital channel wrapped in a uniform, shared, accessible, and protected environment. It is crucial to understand that "uniform" should not be interpreted as monotonous or uncreative; on the contrary, it refers to a consistent and coherent user experience that reduces cognitive load and enhances usability.

Web portals expand on the concept of a traditional website by serving a diverse, segmented audience that includes not only the general public but also internal and external stakeholders like employees, clients, suppliers, governance bodies, and developers. They function as **virtualized companies**, allowing individuals, based on their specific role, to interact with every facet of the organization—from administration and logistics to sales, marketing, human resources, and production. It is a comprehensive platform for:

Multi-Audience Communication : Providing tailored content and functionality for different user groups

Bi-directional Data Interaction : Enabling users to not only consume data but also to input, manage, and interact with it, effectively participating in business processes

Centralized Access : Acting as a single point of access to a wide array of company information, applications, and services

Role-Based Personalization : Ensuring that the experience is secure and customized, granting each user access only to the information and tools relevant to their role

System-to-System Integration : Exposing its functionalities as an API, allowing it to be contacted by other web portals or external systems, which can interact with it programmatically

`WBDL` is the language specifically designed to model the complexity of web portals, defining their structures, data integrations, and authorization rules that govern this dynamic digital ecosystem.

2.3 The Integration Vision

The concept of the ***webbaselet*** opens the door to a vision for the future of enterprise software. In this vision, monolithic and disparate systems like Enterprise Resource Planning (`ERP`), Customer Relationship Management (`CRM`), Business Process Management Systems (`BPMS`), and Manufacturing Resource Planning (`MRP`) no longer need to exist in separate silos with their own disjointed user interfaces.

Instead, the front-end of each of these critical business systems could be engineered as a self-contained ***webbaselet***. These *webbaselet*s could then be seamlessly integrated into a single, unified company ***webbase***.

The result would be a high level of coherence and consistency across the entire enterprise software landscape. Users would navigate a single, familiar portal environment, moving effortlessly between what were once separate applications. This approach would not only improve the user experience but also simplify the development, deployment, and maintenance of enterprise front-ends.

Extending large platforms can be costly. Spin the Web excels at adding smaller, targeted features that are often prohibitively expensive for major vendors to implement.

2.4 The Book Analogy

To better understand the structure of a web portal defined in `WBDL`, it's helpful to use the analogy of a book. The portal is organized hierarchically, much like a book is divided into chapters, pages, and paragraphs. However, the book analogy ends there: web technologies introduce hyperlinks and dynamic routing that open a new, non-linear dimension. The sequence in which users traverse the portal is not fixed; it changes based on navigational setups—menus, breadcrumbs, deep links, search results, role-based visibility, and workflow-driven redirects—so journeys adapt to context, permissions, and intent.

Areas (`STWArea`) : These are the main sections of the portal, analogous to the **chapters** of a book. An area groups related pages together but also other areas.

Pages (`STWPage`) : Contained within Areas, these are the individual **pages** of the book. Each page holds the actual content that users will see.

Content (`STWContent`) : These are the building blocks of a page, analogous to **paragraphs**. They could be tables, trees, menus, tabs, calendars, lists, plots, or any other content that presents data but also interacts with data (APIs); for example, the query of a content, as we'll see later, could be lines of code.

`STWArea`, `STWPage`, and `STWContent` are all specialized types that inherit from the base `STWElement`, sharing its common properties while also having their own specific attributes and behaviors.

2.5 Portal Organization and User Journeys

The structure of a web portal built with `WBDL` is typically organized around the core functions of the business it represents. This creates a logical and intuitive navigation system for all users. Common top-level **Areas (`STWArea`)** would include:

- Sales
- Administration
- Backoffice
- Technical Office
- Logistics
- Products & Services (often publicly viewable)

The full potential of the portal is revealed when we consider the specific journeys of different users, or **personas**. The portal uses a role-based system to present a completely different experience to each user, tailored to their needs and permissions.

2.5.1 Example User Journeys

The Customer:

- Logs into the portal and is directed to a personalized "Customer Dashboard" page

- Can view their complete order history in a dedicated "My Orders" area
- Can track the real-time status of current orders (e.g., "Processing," "Shipped")
- Can initiate a video chat with their designated sales representative directly from the portal
- Can open a support ticket or schedule a consultation with the Technical Office

The Supplier:

- Logs in and sees a "Supplier Dashboard"
- Can access a "Kanban View" page to see which materials or components require replenishment
- Can submit new quotations through an integrated form
- Can view the status of their invoices and payments

The Employee:

- Logs in and is presented with an "Employee Self-Service" area
- Can access a "Welfare Management" page to view and adjust their benefits
- Can view internal company news, submit vacation requests, and access HR documents

The CEO:

- Logs in to a high-level "Executive Dashboard"
- Can view key performance indicators (KPIs) for the entire company, such as sales figures, production output, and financial health
- Can access detailed reports from various departments

These user journeys demonstrate how `WBDL`'s hierarchical structure and role-based visibility rules work together to create a highly functional and personalized web portal that serves as a central hub for the entire business ecosystem.

2.6 The Greater Vision: eBranding Through Portal Development

This book's mission extends beyond technical instruction to address a fundamental challenge in modern web portal development: creating coherent digital brand experiences in an era of fragmented enterprise systems. The Spin the Web approach represents a perspective that organizations can present unified digital identities through thoughtful portal architecture, even when their underlying systems remain disparate. Rather than accepting the status quo of disconnected interfaces that force stakeholders to navigate multiple, inconsistent touchpoints, this framework proposes that web portals can serve as the primary vehicle for eBranding—where an organization's digital presence reflects its true character and values. By embedding business processes, quality management principles, and organizational knowledge directly within portal structures, the framework suggests that portals can evolve from mere functional interfaces into authentic digital representations of the organization itself. This approach to web portal development prioritizes stakeholder experience and organizational coherence, viewing the portal not as a collection of features but as a comprehensive expression of how the organization wishes to engage with its various communities in the digital realm.

2.7 Looking Forward

This introduction has laid out the core problems of enterprise software fragmentation and presented the Spin the Web vision as a solution. But how does this vision translate from an abstract concept into a concrete digital entity? How can a web portal truly become a "virtualized company"?

The next chapter explores this concept in depth, detailing how the project's components work in concert to transform complex business structures into a tangible, unified digital experience.

Chapter 3

Web Portals as Virtualized Companies

"The web of our life is of a mingled yarn, good and ill together."
— William Shakespeare

"The future belongs to organizations that can turn today's information into tomorrow's insight."
— Unknown

The concept of web portals as **virtualized companies** represents a fundamental shift in how we think about enterprise digital presence. Rather than maintaining separate interfaces for different stakeholders, a virtualized company presents a unified, role-based digital environment that encompasses all aspects of business interaction.

3.1 Understanding Virtualized Companies

A virtualized company is more than just a website or even a traditional web portal. It is a comprehensive digital representation of the entire organization, providing **role-specific access** to all business functions, data, and processes through a single, coherent interface.

3.1.1 Key Characteristics

Web portals functioning as virtualized companies exhibit several defining characteristics:

Unified Access Point : All organizational functions are accessible through a single portal, eliminating the need for users to navigate multiple systems or interfaces.

Role-Based Personalization : Each user sees only the information, tools, and functions relevant to their role within the organization, whether they are employees, customers, suppliers, or partners.

Bi-directional Data Flow : Users can both consume and contribute data, participating actively in business processes rather than simply viewing static information.

Dynamic Content Assembly : The portal dynamically assembles content and functionality based on user context, current business state, and real-time data from multiple sources.

Process Integration : Business processes flow seamlessly across traditional departmental boundaries, with the portal orchestrating multi-step workflows that may involve multiple stakeholders.

3.2 The Multi-Audience Challenge

Traditional enterprise software solutions are typically designed for specific user groups or business functions. A virtualized company portal must serve a much more diverse audience:

3.2.1 Internal Stakeholders

Executives and Management : Require high-level dashboards, strategic analytics, and company-wide performance metrics

Department Heads : Need departmental dashboards, resource management tools, and cross-departmental collaboration features

Employees : Access to task management, communication tools, company resources, and role-specific applications

IT and System Administrators : System monitoring, user management, configuration tools, and technical diagnostics

3.2.2 External Stakeholders

Customers : Product catalogs, ordering systems, support portals, account management, and service tracking

Suppliers and Vendors : Supply chain interfaces, order management, quality reporting, and vendor portals

Partners and Distributors : Partner resources, marketing materials, commission tracking, and collaboration tools

Regulatory Bodies : Compliance reporting, audit trails, and required documentation

Investors and Stakeholders : Financial reports, governance information, and strategic communications

3.3 Business Function Integration

A truly virtualized company portal integrates all major business functions into a cohesive whole:

3.3.1 Core Business Areas

Sales and Marketing : Lead management, opportunity tracking, campaign management, customer analytics, and marketing automation

Operations and Production : Manufacturing schedules, quality control, inventory management, and supply chain coordination

Finance and Accounting : Financial reporting, budgeting, expense management, and financial analytics

Human Resources : Employee management, recruitment, performance tracking, and organizational development

Customer Service : Support ticket management, knowledge bases, customer communication, and service analytics

Research and Development : Project management, resource allocation, intellectual property management, and innovation tracking

3.3.2 Cross-Functional Processes

The virtualized company portal excels at supporting processes that span multiple departments:

- **Order-to-Cash**: From initial customer inquiry through order fulfillment and payment processing
- **Procure-to-Pay**: From supplier selection through purchase order management and invoice processing
- **Hire-to-Retire**: Complete employee lifecycle management from recruitment to retirement
- **Idea-to-Market**: Innovation management from concept development through product launch
- **Issue-to-Resolution**: Comprehensive problem management across all business areas

3.4 Technical Architecture Implications

Supporting a virtualized company model requires sophisticated technical architecture:

3.4.1 Data Integration Requirements

- Real-time integration with multiple backend systems
- Data transformation and normalization across different formats
- Master data management to ensure consistency
- Event-driven architecture for real-time updates

3.4.2 Security and Access Control

- Fine-grained role-based access control
- Dynamic permission evaluation
- Audit trails for all user actions
- Data encryption and secure communication

3.4.3 Performance and Scalability

- Efficient caching strategies
- Load balancing across multiple servers
- Asynchronous content loading
- Database optimization and query performance

3.5 Benefits of the Virtualized Company Model

Organizations that successfully implement virtualized company portals typically experience significant benefits:

3.5.1 Operational Efficiency

- Reduced training time for new users
- Faster task completion through unified interfaces
- Elimination of duplicate data entry
- Streamlined approval processes

3.5.2 Improved User Experience

- Single sign-on across all functions
- Consistent user interface and navigation
- Personalized content and functionality
- Mobile-responsive design for anywhere access

3.5.3 Business Intelligence

- Comprehensive analytics across all business functions
- Real-time dashboards and reporting
- Cross-departmental visibility
- Data-driven decision making

3.5.4 Competitive Advantage

- Faster response to market changes
- Improved customer service and satisfaction
- Enhanced partner and supplier relationships
- Greater organizational agility

3.6 Implementation Challenges

While the benefits are significant, implementing a virtualized company portal presents several challenges:

3.6.1 Technical Complexity

- Integration with legacy systems
- Managing diverse data formats and sources
- Ensuring system reliability and uptime
- Maintaining security across all functions

3.6.2 Organizational Change

- Resistance to changing established workflows
- Training requirements across diverse user groups
- Coordinating across multiple departments
- Managing stakeholder expectations

3.6.3 Ongoing Maintenance

- Keeping pace with business changes
- Maintaining data quality and consistency
- Performance optimization and scaling
- Security updates and compliance

3.7 The Spin the Web Approach

Spin the Web addresses these challenges through its unique approach:

`WBDL` Language : Provides a declarative way to define complex portal structures and relationships

Web Spinner Engine : Handles the runtime complexity of role-based content delivery and system integration

Modular Architecture : Enables incremental implementation and easy maintenance

Developer-Focused Tools : Provides professional-grade tools for enterprise developers

3.8 The Portal as the Company's Soul

Ultimately, a web portal built with the Spin the Web philosophy becomes more than just a digital tool; it becomes the natural container for the company's **quality management principles and manuals**. It is the living embodiment of the corporate soul.

By defining not only *what* a company does but also *how* it does it, the portal transforms abstract procedural documents into interactive, enforceable workflows. It ensures that the company's core principles are not just written down, but are actively practiced and experienced by every stakeholder in every interaction. This fusion of documentation and execution is what elevates a web portal from a simple utility to the very heart of the virtualized company.

Chapter 4

Three-Pillar Architecture Overview

"Architecture is about the important stuff. Whatever that is."
— Ralph Johnson

Spin the Web is built upon a core architectural foundation that enables the creation of sophisticated enterprise web portals. This architecture is composed of three distinct but complementary components: the Blueprint, the Engine, and the Workshop. Together, they transform complex business requirements into unified, role-based digital experiences.

4.1 The Core Components

The architecture consists of three core components:

Webbase Description Language (`WBDL`) : A declarative language for modeling complex web portal structures, data relationships, and business logic

Web Spinner Engine : A runtime engine that interprets `WBDL` specifications and delivers dynamic, role-based content

Spin the Web Studio : A specialized *webbaselet* for creating, editing, and managing *webbase* definitions with in-place editing capabilities

4.2 The Blueprint: Webbase Description Language (`WBDL`)

`WBDL` forms the declarative foundation of the entire system. Unlike traditional approaches that mix structure, presentation, and logic, `WBDL` provides a clean separation of concerns through a hierarchical, schema-based approach.

4.2.1 Core Principles

Declarative Definition : Developers describe *what* the portal should do, not *how* it should do it

Data-Driven Structure : Portal elements are defined in terms of their data relationships and business semantics

Role-Based Access : Security and personalization are built into the language specification

Technology Agnostic : `WBDL` abstracts away implementation details, focusing on business requirements

4.2.2 Key Components

`WBDL` defines several fundamental element types:

STWSite : The root element containing global configuration, datasources, and site-wide settings

STWArea : Logical groupings of related functionality, typically corresponding to business domains

STWPage : Individual pages within areas, defining layout sections and content organization

STWContent : Atomic content elements that encapsulate data queries, business logic, and presentation rules

4.2.3 Benefits of the Declarative Approach

- **Maintainability**: Changes to business requirements can be implemented through configuration rather than code changes
- **Consistency**: Standard element types ensure consistent behavior across the entire portal
- **Reusability**: Content elements can be reused across multiple pages and contexts
- **Testability**: Declarative definitions are easier to validate and test than imperative code

4.3 The Engine: Web Spinner

The Web Spinner Engine serves as the runtime foundation that brings `WBDL` specifications to life. It handles the complex orchestration of data retrieval, security enforcement, and content delivery that makes dynamic portals possible.

4.3.1 Core Responsibilities

Configuration Interpretation : Parsing and optimizing `WBDL` documents for runtime execution

Request Routing : Mapping incoming URLs to appropriate portal elements based on the *webbase* structure

Security Enforcement : Implementing role-based access control and content filtering

Data Integration : Orchestrating queries across multiple datasources and systems

Content Assembly : Dynamically composing responses based on user context and permissions

4.3.2 Runtime Architecture

The Web Spinner operates through several key subsystems:

Webbase Loader : Loads and parses `WBDL` documents into optimized in-memory structures

Authorization Engine : Evaluates role-based visibility rules in real-time

Query Processor : Handles `WBPL` placeholder resolution and datasource query execution

Content Cache : Manages caching strategies for improved performance

Session Manager : Maintains user state and authentication information

4.3.3 Performance and Scalability

- **Asynchronous Processing**: Content elements are fetched independently for optimal loading times

- **Intelligent Caching**: Multi-level caching strategies reduce datasource load
- **Connection Pooling**: Efficient resource management for database and API connections
- **Horizontal Scaling**: Support for load-balanced, multi-instance deployments

4.4 The Workshop: Spin the Web Studio

The Spin the Web Studio is a specialized *webbaselet* designed for editing and managing *webbase* definitions. As a *webbaselet*, it can be seamlessly integrated into any *webbase* to provide in-place editing capabilities, enabling professional developers to create and maintain complex portal structures efficiently.

The Studio is dynamically added to the portal *webbase* when users with developer permissions log in, remaining dormant until activated via the **Alt+F12** keyboard shortcut. Upon activation, it transforms the portal interface into a development environment that includes action bars, side panels, status displays, and a main editing area where the live portal remains visible in a persistent browser tab.

4.4.1 Development Capabilities

Visual Design Interface : Graphical tools for designing portal structure and navigation through an interactive *webbase* hierarchy displayed in the side bar

Code Assistance : Intelligent editing support for `WBDL` and `WBPL` syntax with multi-tab file editing capabilities

Data Source Integration : Tools for connecting to and testing various data sources, accessible through portal primary folder contents

Preview and Testing : Real-time preview capabilities with role simulation in the persistent browser tab that displays the live portal

Version Control : Integration with standard source control systems through the side bar source control panel

Comprehensive Search : Unified search capabilities across both *webbase* definitions and folder contents

Integrated Debugging : Built-in debugging tools for `WBPL` queries and *webbaselet* execution

4.4.2 Professional Developer Focus

Unlike low-code/no-code platforms, the Studio is designed specifically for professional developers:

- **Full Control**: Complete access to all `WBDL` features and capabilities
- **Extensibility**: Support for custom components and specialized functionality
- **Integration**: Seamless integration with standard development workflows
- **Performance Tools**: Profiling and optimization capabilities for enterprise-scale deployments

4.5 Architectural Interactions

The Blueprint, Engine, and Workshop work together through well-defined interfaces and protocols:

4.5.1 Design-Time to Runtime

1. Developers use the Studio to create and modify *webbase* definitions
2. `WBDL` documents are saved and versioned using standard development practices
3. The Web Spinner Engine loads updated *webbase* definitions and applies changes
4. Changes take effect immediately without requiring system restarts

4.5.2 Runtime Operations

1. User requests arrive at the Web Spinner Engine
2. The engine consults the in-memory *webbase* structure to route the request
3. Authorization rules defined in `WBDL` are evaluated against user roles
4. Appropriate `STWContent` elements are identified and executed
5. Responses are assembled and delivered to the client

4.5.3 Feedback Loops

- Runtime performance metrics inform Studio optimization recommendations
- User behavior analytics guide design decisions
- Error logs and debugging information flow back to the development environment
- A/B testing capabilities enable data-driven design refinements

4.6 Architectural Benefits

This architectural model provides several key advantages over traditional portal development methods:

4.6.1 Separation of Concerns

- Business logic is separated from presentation concerns
- Security policies are defined declaratively rather than embedded in code
- Data access patterns are standardized and centrally managed
- UI rendering is handled consistently across all portal components

4.6.2 Development Efficiency

- Faster development cycles through declarative configuration
- Reduced debugging time due to standardized runtime behavior
- Easier maintenance through centralized configuration management
- Lower training requirements for new team members

4.6.3 Enterprise Scalability

- Proven performance characteristics for large-scale deployments
- Built-in support for high availability and disaster recovery
- Comprehensive monitoring and operational management capabilities
- Integration with enterprise identity and security systems

4.7 Implementation Patterns

Common implementation patterns emerge when working with this architecture:

4.7.1 Incremental Development

1. Start with core site structure and basic navigation
2. Add key business areas one at a time
3. Implement critical content elements first
4. Gradually add advanced features like real-time updates and complex workflows

4.7.2 Modular Design

- Design `STWContent` elements as reusable components
- Create area-specific templates for consistent user experience
- Develop shared datasource configurations for common data patterns
- Build libraries of common UI components and layouts

4.7.3 Testing and Validation

- Use Studio preview capabilities for immediate design feedback
- Implement automated testing for critical business logic
- Perform role-based testing to validate security configurations
- Conduct performance testing under realistic load conditions

4.8 Looking Forward

With the high-level architecture of the Blueprint, Engine, and Workshop defined, our focus must now turn to the underlying engineering. How are these components built? What are the technical specifications of the languages and the internal mechanics of the runtime?

The next part of this book is dedicated to a deep dive into the framework's implementation—the platform itself. We will explore the complete specifications of the languages and the architectural details of the Web Spinner engine.

Part II

The Platform

Introduction to Part II: The Platform

> *"The limits of my language mean the limits of my world."*
> — Ludwig Wittgenstein

This part opens the hood of the Spin the Web framework to provide the complete technical specification for "the platform" itself. We will dissect the core components introduced in the architecture overview: the languages that serve as the platform's instruction set, the modular design that enables scalability, and the mechanics of the engine that brings it all to life.

Mastering these specifications is the first step toward understanding how to build, operate, and extend the framework. This section serves as the definitive engineering blueprint for the Spin the Web ecosystem.

Chapter 5: The WBDL Language (chapter 5) – Provides a comprehensive introduction to the Webbase Description Language, including its JSON Schema definitions, element hierarchy, and practical usage patterns.

Chapter 6: The WBPL Language (chapter 6) – Explores the Webbase Placeholders Language, which enables dynamic content injection and template-based portal generation.

Chapter 7: The WBLL Language (chapter 7) – Defines the Webbase Layout Language, presentation layer, tokens, helpers, and compilation model used to render data into accessible, responsive HTML.

Chapter 8: Webbase and Webbaselets (chapter 8) – Examines the modular component system that allows for reusable portal elements and cross-platform integration.

Chapter 9: The Web Spinner Engine (chapter 9) – Details the runtime engine that processes WBDL specifications and generates dynamic web portals, including its architecture, processing pipeline, and performance characteristics.

Chapter 10: The Spin the Web Studio (chapter 10) – Introduces the integrated development environment for building and managing webbases.

Chapter 11: Reference Implementation (chapter 11) – Documents the concrete technology stack (Deno/TypeScript) of the reference Web Spinner, showing how the theoretical mechanics are realized in a working system.

Mastering these specifications is the first step toward building robust, enterprise-grade portals. This section serves as the definitive reference for the framework's "Blueprint," "Engine," and "Workshop."

Chapter 5

WBDL Language

> *"The limits of my language mean the limits of my world."*
> — Ludwig Wittgenstein

`WBDL` is formally defined using the standard **JSON Schema**.

5.1 WBDL Ecosystem

While `WBDL` serves as the declarative language for describing web portals, it operates within a comprehensive ecosystem of associated technologies and APIs that enhance its capabilities:

5.1.1 Layout API

`WBDL` includes an associated Layout API that provides programmatic control over content presentation and styling. This API works in conjunction with the declarative `STWLayout` elements to enable:

- Dynamic layout adjustments based on content type and user preferences
- Responsive design patterns that adapt to different screen sizes and devices
- Integration with external CSS frameworks and design systems
- Runtime modification of layout properties based on data characteristics

The Layout API ensures that content presentation remains consistent across different contexts while allowing for necessary flexibility in complex enterprise environments.

5.1.2 Text Preprocessor

The `WBDL` text preprocessor is a powerful engine that processes textual content before rendering, enabling:

- Placeholder substitution using the **Webbase Placeholders Language (`WBPL`)**
- Localization and internationalization support
- Dynamic text generation based on context and user data
- Integration with external content management systems
- Markdown and other markup language processing

This preprocessor is particularly crucial for the `query` attributes in `STWContent` elements, where it replaces placeholders with actual values from various sources before query execution.

5.1.3 Visual Component Library (VCL)

The Visual Component Library provides a standardized set of UI components that correspond to the different content categories and subtypes. The VCL ensures:

- Consistent visual presentation across different portal implementations
- Accessibility compliance through standardized component behavior
- Cross-platform compatibility for various front-end frameworks
- Extensibility for custom component development

5.1.4 Content Management Integration

`WBDL` is designed as a fundamental language for Content Management Systems (CMS), providing:

- Structured content definition that separates presentation from data
- Version control capabilities for portal configurations
- Workflow management for content approval processes
- Integration points for external content repositories

This integration capability makes `WBDL` particularly suitable for enterprise environments where content management workflows are critical to business operations.

5.2 Spin the Web Studio

The third and final component of Spin the Web is the **Spin the Web Studio**. Alongside the **`WBDL`** and the **Web Spinner**, it completes the framework. The Spin the Web Studio is a specialized *webbaselet* engineered for editing *webbase*s. To use it, you simply add the *webbaselet* to the *webbase* you wish to edit, enabling direct, in-place modification.

5.3 The STWElement Base

`WBDL` defines a base element, `STWElement`, from which all other elements inherit. Below is the JSON Schema definition for this fundamental type.

```
{
  "$id": "https://spintheweb.org/schemas/STWElement.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWElement",
  "description": "The base element for all WBDL objects.",
  "type": "object",
  "properties": {
    "_id": { "type": "string", "format": "uuid" },
    "type": { "enum": ["Site", "Area", "Page", "Content"] },
    "name": { "$ref": "STWLocalized.json" },
    "slug": { "$ref": "STWLocalized.json" },
```

```
    "keywords": { "$ref": "STWLocalized.json" },
    "description": { "$ref": "STWLocalized.json" },
    "visibility": { "$ref": "STWVisibility.json" },
    "children": {
      "type": "array",
      "items": { "$ref": "#" }
    }
  },
  "required": ["_id", "type", "name", "slug"]
}
```

Listing 5.1: STWElement Base Schema Definition

5.3.1 Property Description

_id : A unique identifier for the element (GUID).

type : The specific type of the element. Can be one of 'Site', 'Area', 'Page', or 'Content'.

name : A localizable name for the element. This is a JSON object where keys are language codes (e.g., "en", "it") and values are the translated strings. Example: `{"en": "Research & Development", "it": "Ricerca e Sviluppo"}`

slug : A localizable, URL-friendly version of the name. It is a JSON object with the same structure as `name`. Slugs should contain only lowercase letters, numbers, and hyphens.

keywords : Localizable keywords for SEO. Follows the same structure as `name`.

description : A localizable description of the element. Follows the same structure as `name`.

visibility : Defines the visibility rules for the element based on user roles. It contains a set of rules, where each rule assigns `true` (visible) or `false` (not visible) to a specific role. If a rule for a role is not defined (is `null`), the visibility is determined by checking the parent element's visibility rules. This hierarchical check continues up to the root element. If no rule is found, the element is not visible by default.

children : A list of child `STWElement` objects.

5.4 WBDL Element Types

This section describes the specialized element types available in `WBDL`.

5.4.1 STWSite

`STWSite` is a singleton element that represents the entire web portal. It inherits from `STWElement` and acts as the root element of the portal structure.

```
{
  "$id": "https://spintheweb.org/schemas/STWSite.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWSite",
  "allOf": [{ "$ref": "STWElement.json" }],
  "properties": {
```

```
    "type": { "const": "Site" },
    "mainpage": { "type": "string", "format": "uuid" },
    "langs": {
      "type": "array",
      "items": { "type": "string" }
    },
    "datasources": {
      "type": "array",
      "items": { "type": "object" }
    },
    "children": {
      "type": "array",
      "items": { "$ref": "STWArea.json" }
    }
  },
  "required": ["mainpage"]
}
```

Listing 5.2: STWSite Schema Definition

Property Description

langs : A list of supported languages for the site. The first of the list is the default site language.
datasources : Defines the data sources used by the portal.
mainpage : The GUID of the `STWPage` that serves as the main entry point for the site.
version : A version string for the site.

Data Sources and Groups

The `datasources` element within `STWSite` provides a flexible framework for defining how the portal connects to and manages external data systems. Beyond simple connection definitions, `WBDL` supports advanced data organization concepts:

Data Source Configuration : Each data source can be configured with specific connection parameters, authentication credentials, and query optimization settings. Supported data source types include:

- Relational databases (SQL-based)
- NoSQL databases (document, key-value, graph)
- REST APIs and web services
- File-based data sources (CSV, JSON, XML)
- Enterprise systems (ERP, CRM, HRM)

Data Groups : Data sources can be organized into logical groups that facilitate:

- Access control and security policies
- Performance optimization through connection pooling
- Load balancing across redundant data sources
- Backup and failover configurations
- Administrative grouping for different business units

Accessibility Integration : The data source framework includes provisions for accessibility compliance:

- Metadata enrichment for screen readers and assistive technologies
- Alternative data representations for different accessibility needs
- Query result formatting that supports accessibility standards
- Integration with accessibility frameworks and tools

This comprehensive approach to data source management ensures that `WBDL` can integrate seamlessly with complex enterprise data environments while maintaining accessibility and security standards.

5.4.2 STWArea

`STWArea` represents a logical grouping of pages, analogous to a chapter in a book. It extends the base `STWElement`.

```
{
  "$id": "https://spintheweb.org/schemas/STWArea.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWArea",
  "allOf": [{ "$ref": "STWElement.json" }],
  "properties": {
    "type": { "const": "Area" },
    "mainpage": { "type": "string", "format": "uuid" }
  },
  "required": ["mainpage"]
}
```

Listing 5.3: STWArea Schema Definition

Property Description

mainpage : The GUID of the `STWPage` that serves as the main entry point for this area.
version : A version string for the area.

5.4.3 STWPage

`STWPage` represents a single page in the portal. It extends the base `STWElement` but restricts its `children` to only `STWContent` elements.

```
{
  "$id": "https://spintheweb.org/schemas/STWPage.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWPage",
  "allOf": [{ "$ref": "STWElement.json" }],
  "properties": {
    "type": { "const": "Page" },
    "children": {
      "type": "array",
```

```
      "items": { "$ref": "STWContent.json" }
    }
  },
  "required": ["children"]
}
```

Listing 5.4: STWPage Schema Definition

5.4.4 STWContent

`STWContent` represents a piece of content on a page. It extends the base `STWElement` but is not allowed to have any `children`. It adds several attributes for data binding and layout control.

```
{
  "$id": "https://spintheweb.org/schemas/STWContent.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWContent",
  "allOf": [{ "$ref": "STWElement.json" }],
  "properties": {
    "type": { "const": "Content" },
    "subtype": { "type": "string" },
    "cssClass": { "type": "string" },
    "section": { "type": "string" },
    "sequence": { "type": "integer" },
    "dsn": { "type": "string" },
    "query": { "type": "string" },
    "params": { "type": "string" },
    "layout": { "$ref": "STWLayout.json" }
  },
  "required": ["layout"]
}
```

Listing 5.5: STWContent Schema Definition

Property Description

type : Overrides the base element's type and is fixed to "Content".

subtype : Specifies the type of content to be rendered, which determines the component used on the front-end. Possible values include `Text`, `Form`, `Table`, `Tree`, `Calendar`, and `Breadcrumbs`.

cssClass : An optional CSS class to apply to the content element for styling.

section : The name of the page section where this content should be rendered (e.g., "header", "main", "sidebar").

sequence : A number that determines the order of content within a section.

dsn : The "data source name," which identifies a specific data source configured in the `STWSite` element.

query : The query to be executed against the specified data source. Before execution, the query text is processed by the **Webbase Placeholders Language (`WBPL`)** processor. This processor replaces placeholders within the query with values sourced from several locations: the `params` attribute, the URL querystring, session variables, global variables, and HTTP headers. After this substitution, the resulting query is executed by the data source using its native language (e.g., SQL for a relational database, or JSONata if the data source is a JSON-based API).

params : A string containing parameters for the query, formatted as a standard query string (e.g., `key1=value1&key2=value2`).

layout : The `STWLayout` element that defines how the fetched data should be rendered.

Content Categories

Contents are the fundamental elements of `WBDL` and represent interactive data units that fall into four distinct categories, each serving a specific purpose in the portal's user interface:

Sensorial Contents : These render data in human-readable formats for information consumption and input. They include:

- Free text displays
- Forms for data input
- Lists and tables for structured data presentation
- Plots and charts for data visualization
- Maps for geographical data
- Timelines for temporal data
- Media elements (images, videos, audio)

Navigational Contents : These render data as interactive links and navigation elements, facilitating movement through the portal structure:

- Menus (primary and secondary navigation)
- Table of contents (TOC)
- Breadcrumbs for hierarchical navigation
- Slicers for data filtering
- Image maps with clickable regions
- Pagination controls

Organizational Contents : These wrap and organize other contents in structured manners, providing logical grouping and presentation:

- Tabs for content organization
- Calendars for date-based content
- Trees for hierarchical data
- Graphs for relationship visualization
- Accordions for collapsible sections
- Carousels for sequential content display

Special Contents : These provide specialized functionality and shortcuts:

- Shortcuts to frequently accessed content
- Code displays with syntax highlighting

- Embedded widgets and components
- Custom functionality modules

All content types explicitly declare or request the data they interact with, and their rendering behavior is determined by their category and specific subtype. The content categorization ensures consistency in user interface patterns and enables the Web Spinner to apply appropriate rendering logic and security policies.

5.4.5 STWContentWithOptions

`STWContentWithOptions` is similar to `STWContent`, it contains a list of `option` elements (GUIDs). This type is useful for scenarios like menus and tabs where the content is sourced from other elements.

```
{
  "$id": "https://spintheweb.org/schemas/STWContentWithOptions.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWContentWithOptions",
  "allOf": [{ "$ref": "STWElement.json" }],
  "properties": {
    "type": { "const": "Content" },
    "subtype": { "type": "string" },
    "options": {
      "type": "array",
      "items": { "type": "string", "format": "uuid" }
    },
    "layout": { "$ref": "STWLayout.json" }
  },
  "required": ["options"]
}
```

Listing 5.6: STWContentWithOptions Schema Definition

Property Description

subtype : Specifies the type of content to be rendered. Possible values include `Menu`, `Tabs`, and `Accordion`.

5.5 Looking Forward

A static portal, however well-structured, has limited utility in a dynamic business environment. The true power of an enterprise portal lies in its ability to present real-time, contextualized data. But how do we bridge the gap between the static structure of `WBDL` and the dynamic data living in enterprise systems?

The next chapter introduces the **Webbase Placeholders Language (`WBPL`)**, a specialized language designed to embed dynamic queries and data-driven logic directly within your `WBDL` definitions. We will explore how `WBPL` turns your static templates into living, breathing documents.

Chapter 6

Webbase Placeholders Language (WBPL)

"The real problem is not whether machines think but whether men do."
— B.F. Skinner

The Webbase Placeholders Language (`WBPL`) is a string processing language designed for creating dynamic, data-driven queries. It works by taking a template string and a map of placeholders, then producing a final string by substituting placeholders and conditionally including or excluding parts of the template based on whether the placeholders have values.

6.1 Core Functionality

Here is a detailed breakdown of `WBPL` functionality:

6.1.1 Placeholder Syntax

`WBPL` identifies placeholders using an `@` symbol. The syntax supports different forms, such as `@`, `@@`, and `@@@`, which may have distinct meanings. Placeholders can be used directly (unquoted) or enclosed in single or double quotes.

6.1.2 Substitution Mechanisms

Simple Substitution : For unquoted placeholders like `@name`, the engine directly replaces them with the corresponding value from the placeholders map.

Quoted Substitution : For quoted placeholders like `'@name'`, the engine replaces them with the value, automatically ensuring the value is correctly quoted and that any internal quotes are escaped. This is crucial for safely embedding string values in languages like SQL.

List Expansion : A special syntax exists for quoted placeholders followed by an ellipsis (...), such as `'@ids'...'`. This is designed to expand a comma-separated value into a properly quoted, comma-separated list (e.g., expanding a string `"1,2,3"` into `'1','2','3'`). This is particularly useful for generating SQL `IN` clauses.

6.1.3 Conditional Blocks

The language supports two types of conditional blocks that control whether a piece of text is included in the final output.

Curly Braces {...}

Text inside curly braces is included **only if** at least one placeholder within it is successfully replaced with a non-empty value. If all placeholders inside are empty or non-existent, the entire block (including the braces) is removed. This is useful for including optional text that depends on a value being present.

Square Brackets [...]

This block behaves similarly to curly braces, but with an added feature for cleaning up query syntax. If the block is removed because its placeholders are empty, the engine will also intelligently remove a single adjacent keyword (like `AND`, `OR`, `WHERE`). This is designed to handle optional conditional clauses in SQL.

For example, in `SELECT * FROM users WHERE 1=1 [AND user_id = @id]`, the `AND` keyword and the entire bracketed expression will be removed if `@id` has no value.

6.1.4 Escaping

- The language respects escaped at-symbols (`\@`), treating them as literal `@` characters rather than the start of a placeholder.
- It correctly handles and escapes quotes within values during substitution to prevent syntax errors or potential injection vulnerabilities.

6.2 Security Features

In essence, `WBPL` is a security-conscious templating engine tailored for generating dynamic queries and other text formats where parts of the content are conditional. Key security features include:

- Automatic quote escaping to prevent SQL injection attacks
- Input validation and sanitization
- Safe handling of user-provided parameters
- Proper encoding for different target languages (SQL, JSON, etc.)

6.3 Usage Examples

6.3.1 Basic Placeholder Substitution

```
SELECT * FROM users WHERE username = '@username'
```

Listing 6.1: Simple WBPL Substitution

With placeholder `username = "john_doe"`, this becomes:

```
SELECT * FROM users WHERE username = 'john_doe'
```

6.3.2 Conditional Query Clauses

```
SELECT * FROM products
WHERE 1=1
[AND category = '@category']
[AND price >= @min_price]
[AND price <= @max_price]
```

Listing 6.2: WBPL Conditional Blocks

If only `category = "electronics"` is provided, this becomes:

```
SELECT * FROM products
WHERE 1=1
AND category = 'electronics'
```

6.3.3 List Expansion for IN Clauses

```
SELECT * FROM orders WHERE status IN ('@statuses'...)
```

Listing 6.3: WBPL List Expansion

With placeholder `statuses = "pending,shipped,delivered"`, this becomes:

```
SELECT * FROM orders WHERE status IN ('pending','shipped','delivered')
```

6.4 Integration with WBDL

`WBPL` is primarily used within `STWContent` elements in `WBDL` documents. The `query` attribute of an `STWContent` element contains a template string that is processed by the `WBPL` engine before being executed against the specified datasource.

The placeholder values are sourced from multiple locations in order of precedence:

1. The `params` attribute of the `STWContent` element
2. URL query string parameters
3. Session variables (user context, roles, preferences)
4. Global variables (site configuration, system settings)
5. HTTP headers (for device type, language preferences, etc.)

6.5 Performance Considerations

The WBPL processor is optimized for high-performance scenarios:

- Template parsing is cached to avoid repeated compilation
- Placeholder resolution is optimized for minimal overhead
- Query plans may be cached when placeholder patterns are stable
- Security validation is performed efficiently without sacrificing safety

6.6 Looking Forward

While WBDL provides the structure and WBPL injects dynamic data, a critical question remains: how do we manage the complexity of a large-scale enterprise portal? How do we ensure that different parts of the portal can be developed, maintained, and deployed independently without causing system-wide disruptions?

The next chapter introduces the concepts of ***webbases*** and ***webbaselets***—the architectural solution for modularity and scalability in the Spin the Web ecosystem. We will see how these concepts allow for a "divide and conquer" approach to portal development.

Chapter 7

Webbase Layout Language (WBLL)

"Simplicity is the ultimate sophistication."
— Leonardo da Vinci

In `WBDL`, the portal is specified *macroscopically*: from *Site* to *Area* to *Page*, and for each Page the set of *Contents*. For every Content, `WBDL` declares its *subtype* (for example: table, calendar, tree, menu, form), the *datasource*, and the *query* to run. WBLL operates *microscopically*, describing how the queried data should be rendered within that Content's shape.

7.0.1 Data Context and Field Cursor

It is the responsibility of the `WBDL` content definition to query data. The result is an ordered recordset, where both the field names and their sequence are important for rendering.

The WBLL interpreter maintains a **field cursor** that tracks the current position within the recordset fields. Many tokens operate on the field at the current cursor position if no explicit field name is provided; tokens that specify a field name use that field without moving the cursor. Some tokens may also vary their behavior based on the content subtype; this is documented with each token where relevant.

Cursor movement rule By convention, the field cursor advances if and only if a token implicitly consumes the active field (i.e., it reads a field value without naming a field explicitly). Tokens that only render constants, or that reference a field by name, do not move the cursor.

- Cursor-advancing (implicit field): a token used without a field name consumes the active field and advances the cursor.
  Example: `h` renders `<input type="hidden" name="<active>" value="<value>">` and advances.
- Non-advancing (explicit field): a token that names a field uses that field but leaves the cursor unchanged.
  Example: `h('type;area')` renders `<input type="hidden" name="type" value="area">` and does not advance.
- Non-consuming: tokens that render only literals or structural markup do not touch the cursor.
  Example: `t('Hello World!')` does not advance.

Implementations typically mark a token as having consumed the active field only if it bound to it implicitly; explicit bindings and literal-only tokens leave the cursor position intact.

This chapter establishes the presentation-layer contracts. In the next chapters we detail modular composition (chapter 8) and the runtime engine (chapter 9).

7.0.2 Language Internals

At its core, WBLL is a compact, token-based templating language. Unlike tag-based languages like HTML, WBLL uses a sequence of single-character mnemonics and special commands to define the structure and appearance of content. This design prioritizes conciseness and efficient processing.

The rendering process involves two main stages:

1. **Lexing (Tokenization):** The WBLL source string is first parsed by a lexer, which uses a comprehensive regular expression to break the text into a series of tokens. Each token represents a specific layout element, such as a link (`a`), a button (`b`), a form input (`e`), or a simple text block (`t`). The lexer also captures arguments, attributes, and parameters associated with each token. Any unrecognized characters result in a syntax error.
2. **Compilation and Rendering:** The resulting array of tokens is then compiled into a dynamic JavaScript render function. This function is specifically generated to produce HTML from the token sequence. When executed, it iterates through the tokens, merges data from records and session placeholders, and constructs the final HTML string. This just-in-time compilation allows for highly efficient rendering, as the logic is tailored precisely to the given layout.

7.0.3 Token Reference

For the complete, uniform catalog of tokens (syntax, description, and examples), see Appendix A.

Chapter 8

Webbase and Webbaselets

> *"The whole is more than the sum of its parts."*
> — Aristotle

A complete `WBDL` document, representing a full portal, is called a ***webbase***. A key requirement for a valid *webbase* is that it must contain exactly one `STWSite` element, which serves as the root of the entire structure.

However, it is also possible to create smaller, modular `WBDL` files called ***webbaselet*s**. A *webbaselet* is a `WBDL` document that does *not* contain an `STWSite` element. Instead, its root element must be an `STWArea`. *webbaselet*s are designed to be portable fragments that can be imported or included within a larger *webbase*.

This modularity ensures that the portal can evolve without requiring a monolithic update, promoting agility and long-term maintainability.

8.1 Webbase Structure

A *webbase* represents a complete, self-contained web portal. It includes:

Site Configuration : The root `STWSite` element containing global settings, supported languages, and datasource definitions

Navigation Hierarchy : A complete tree of areas, pages, and content elements that define the portal structure

Security Model : Comprehensive visibility rules and role-based access controls throughout the hierarchy

Data Integration : Datasource configurations and query templates for dynamic content generation

8.1.1 Webbase Example Structure

```
{
  "_id": "12345678-1234-1234-1234-123456789012",
  "type": "Site",
  "version": "1.0",
  "mainpage": "87654321-4321-4321-4321-210987654321",
```

```
  "name": { "en": "Corporate Portal" },
  "slug": { "en": "portal" },
  "langs": ["en", "it", "fr"],
  "datasources": [
    { "name": "main", "type": "postgresql", "connectionString": "..." }
  ],
  "children": [
    {
      "_id": "...",
      "type": "Area"
    }
  ]
}
```

Listing 8.1: Basic Webbase Structure in JSON

8.1.2 Webbaselet Definition

A *webbaselet* is a valid `WBDL` document whose root element is an `STWArea`.

```
{
  "_id": "hr-webbaselet-root",
  "type": "Area",
  "name": { "en": "Human Resources" },
  "slug": { "en": "human-resources" },
  "children": [
    {
      "_id": "hr-page-1",
      "type": "Page",
      "name": { "en": "Employee Directory" },
      "slug": { "en": "directory" }
    }
  ]
}
```

Listing 8.2: Example of a Webbaselet in JSON

8.1.3 Namespace Isolation

Each *webbaselet* can define its own namespace by adding a `namespace` property to the root `STWArea` element.

```
{
  "_id": "hr-webbaselet-root",
  "type": "Area",
  "namespace": "hr",
  "name": { "en": "Human Resources" },
  "slug": { "en": "human-resources" }
```

```
}
```

Listing 8.3: Webbaselet with Namespace

When this *webbaselet* is integrated, the final URL for the page could be resolved to `/human-resources/directory`, with the engine managing any potential conflicts.

8.2 Development Workflow

The *webbase*/*webbaselet* architecture enables sophisticated development workflows:

8.2.1 Modular Development

Different teams can work on separate *webbaselet*s simultaneously:

- HR team develops employee self-service *webbaselet*
- Sales team develops customer portal *webbaselet*
- IT team develops system administration *webbaselet*
- Each team can test independently before integration

8.2.2 Versioning and Release Management

*webbaselet*s support independent versioning:

- Each *webbaselet* maintains its own version number
- Compatible versions can be mixed within a single *webbase*
- Rollback capabilities for individual *webbaselet*s
- A/B testing of different *webbaselet* versions

8.2.3 Testing and Quality Assurance

Modular architecture improves testing:

- Unit testing of individual *webbaselet*s
- Integration testing of *webbaselet* combinations
- Isolated regression testing when updating specific *webbaselet*s
- Performance testing of high-traffic *webbaselet*s

8.3 Governance and Security

The modular architecture supports enterprise governance requirements:

8.3.1 Access Control

*webbaselet*s can implement their own security models:

- Role-based permissions specific to *webbaselet* functionality
- Integration with enterprise identity providers
- Audit trails for *webbaselet*-specific actions

- Data loss prevention for sensitive *webbaselet*s

8.3.2 Compliance

Different *webbaselet*s can meet different compliance requirements:

- GDPR compliance for EU customer data *webbaselet*s
- SOX compliance for financial reporting *webbaselet*s
- HIPAA compliance for healthcare-related *webbaselet*s
- Industry-specific regulations for specialized *webbaselet*s

8.3.3 Change Management

webbaselet deployment can be controlled through governance processes:

- Approval workflows for *webbaselet* updates
- Automated testing before *webbaselet* deployment
- Rollback procedures for problematic *webbaselet*s
- Change impact analysis for *webbaselet* modifications

8.4 Performance and Scalability

The modular architecture provides performance benefits:

8.4.1 Selective Loading

The Web Spinner can optimize performance by:

- Loading only *webbaselet*s relevant to the current user's roles
- Lazy loading of *webbaselet*s when first accessed
- Caching frequently used *webbaselet*s in memory
- Unloading unused *webbaselet*s to conserve resources

8.4.2 Distributed Deployment

*webbaselet*s can be deployed across multiple servers:

- High-traffic *webbaselet*s on dedicated servers
- Geographic distribution of region-specific *webbaselet*s
- Load balancing based on *webbaselet* usage patterns
- Failover capabilities for critical *webbaselet*s

8.5 Future Evolution

The *webbase*/*webbaselet* architecture is designed to support future enhancements:

8.5.1 Marketplace Integration

A future ecosystem could include:

- Third-party *webbaselet* marketplace
- Certified *webbaselet*s from trusted vendors
- Community-contributed open-source *webbaselet*s
- Enterprise *webbaselet* repositories

8.5.2 AI-Driven Development

Future tools could provide:

- Automated *webbaselet* generation from requirements
- Intelligent *webbaselet* recommendations based on usage patterns
- Automatic optimization of *webbaselet* performance
- Predictive analytics for *webbaselet* maintenance

8.6 Looking Forward

We have now defined the languages (`WBDL`, `WBPL`, `WBLL`) and the modular architecture (*webbaselet*s) that form the static blueprint of a Spin the Web portal. But how is this blueprint brought to life? What is the machinery that parses these definitions, executes queries, enforces security, and renders the final user interface?

The next chapter provides a detailed look at the ***Web Spinner* Engine**, the high-performance runtime that serves as the heart of the Spin the Web project. We will explore its internal architecture, from request handling to final HTML rendering.

Chapter 9

Web Spinner Engine Architecture and Mechanics

"The web of our life is of a mingled yarn, good and ill together."
— William Shakespeare

`WBDL` is processed by a server-side engine called the **Web Spinner**. In a direct analogy to how a client-side web browser renders HTML into a user-facing webpage, the Web Spinner interprets `WBDL` descriptions to generate and manage the web portal on the server. It is the runtime engine of the project, complemented by the **Spin the Web Studio**, which serves as the design-time tool for creating and modifying the *webbase* itself. This architecture allows for dynamic, data-driven portal generation based on the `WBDL` specification.

When the Web Spinner starts, it loads the *webbase* (in JSON format) into memory. This *webbase* is transformed into an optimized, in-memory object, allowing for fast access and manipulation of the portal's structure.

9.1 Primary Roles and Responsibilities

The Web Spinner's primary roles are routing and content delivery:

1. **URL-Based Routing:** It maps the incoming URL directly to a *webbase* object. For example, a URL like `https://portal.acme.com/areaslug/areaslug/pageslug` is interpreted as a path to a specific `STWPage` element within the *webbase* hierarchy.
2. **Page Composition:** When a user requests a page, the Web Spinner acts as the initial router. It identifies the requested page and its associated `STWContent` elements. Crucially, it checks the `visibility` rules for the page and each content element against the user's roles. Elements that are not visible to the user are filtered out and never sent to the client. The spinner then returns the list of slugs for the visible contents.
3. **Asynchronous Content Fetching:** The client receives the list of content slugs and proceeds to request each one asynchronously and individually. For instance, it would request `https://portal.acme.com/areaslug/areaslug/pageslug/contentslug`. This approach allows the main page structure to load quickly while content is fetched in parallel, improving perceived performance.

4. **Dynamic Content:** A content element is not limited to rendering data but also managing data like an API.

This architecture ensures that the server handles the core logic of structure, routing, and security, while the client is responsible for the final rendering, leading to a flexible and performant web portal.

9.2 System Startup and Initialization

When the Web Spinner starts up, it performs several critical initialization steps:

9.2.1 Webbase Loading

The engine loads the complete `WBDL` document (*webbase*) from disk, stored in JSON format. This document contains the entire portal structure, including all areas, pages, content definitions, and configuration data.

9.2.2 In-Memory Optimization

The loaded *webbase* is parsed and transformed into an optimized in-memory object graph. This transformation includes:

- Resolving all GUID references between elements
- Building navigation hierarchies for fast traversal
- Pre-compiling visibility rules for rapid role-based filtering
- Indexing elements by slug for efficient URL routing

9.2.3 Datasource Configuration

All datasources defined in the `STWSite` element are initialized and connection pools are established. This includes databases, `REST` APIs, file systems, and any other external data providers.

9.2.4 Route Table Generation

A comprehensive routing table is built from the *webbase* structure, mapping URL patterns to specific `STWPage` and `STWContent` elements.

9.3 User Session Management

The Web Spinner maintains stateful user sessions to manage authentication, authorization, and personalization:

9.3.1 Session Establishment

When a user first accesses the portal, a new session is created with a unique identifier, guest is set as the session user (guest has is assigned the guests role). Sessions are maintained using cookies, JWT tokens, or other mechanisms.

9.3.2 Authentication Integration

The system integrates with various authentication providers (LDAP, OAuth, SAML, etc.) to verify user credentials and establish their identity.

9.3.3 Role Assignment

Once authenticated, the user's roles are determined through integration with identity providers or internal role mappings. These roles are cached in the session for performance.

9.3.4 Session State

The session maintains:

- User identity and roles
- Current language preference
- Navigation history
- Cached query results (when appropriate)
- Active datasource connections

9.4 Request Routing and Processing

The Web Spinner handles incoming HTTP requests through a sophisticated routing mechanism:

9.4.1 URL Parsing

Incoming URLs are parsed to extract the area/page/content hierarchy. For example:

- `https://portal.acme.com/sales/dashboard` → Area: "sales", Page: "dashboard"
- `https://portal.acme.com/sales/dashboard/orders-table` → Area: "sales", Page: "dashboard", Content: "orders-table"

9.4.2 Element Resolution

Using the pre-built routing table, URLs are resolved to specific `WBDL` elements. Missing or invalid paths result in no result in case of contents the nearest site or area main page in case of a page.

9.4.3 Protocol Handling

The Web Spinner supports both HTTP and WebSocket protocols:

HTTP : Used for standard page and content requests, following RESTful principles
WebSocket : Used for real-time content updates, live data feeds, and interactive features

9.5 Visibility and Authorization Engine

Before any content is delivered, the Web Spinner performs comprehensive authorization checks:

9.5.1 Role-Based Filtering

For each requested element (page or content), the visibility rules are evaluated against the user's session roles:

- Elements with no matching role rules inherit visibility from their parent elements
- The hierarchical check continues up to the root element
- Elements without explicit or inherited visibility permissions are denied by default

9.5.2 Dynamic Filtering

Visibility checks are performed in real-time for every request, ensuring that changes to user roles or permissions take immediate effect.

9.5.3 Secure Response Generation

Only authorized elements are included in the response. Unauthorized elements are completely omitted, preventing information leakage.

9.6 Content Request and Response Cycle

The Web Spinner handles content requests through a multi-stage process:

9.6.1 Page Structure Delivery

When a page is requested, the Web Spinner:

- Identifies all visible `STWContent` elements for the page
- Returns a lightweight response containing the list of content slugs and their metadata
- Includes section assignments and sequencing information for layout

9.6.2 Asynchronous Content Fetching

The client then requests individual content elements:

- Each content request triggers a separate web socket call to the Web Spinner
- Content elements are fetched in parallel, improving perceived performance
- Each content element is processed independently, allowing for granular caching and error handling

9.6.3 Content Processing Pipeline

For each content request, the Web Spinner:

1. Validates user authorization for the specific content element
2. Resolves the associated datasource and query
3. Processes the query through the `WBPL` (Webbase Placeholders Language) engine
4. Executes the processed query against the datasource
5. Applies the content layout transformation
6. Returns the rendered content or raw data (depending on the request type)

9.7 Datasource Connection and Query Management

The Web Spinner maintains a sophisticated datasource management system:

9.7.1 Connection Pooling

Database and API connections are pooled and reused across requests to optimize performance and resource utilization.

9.7.2 Query Processing

Raw queries defined in `STWContent` elements undergo several processing steps:

`WBPL` Processing : Placeholders are resolved using session data, URL parameters, and global variables

Security Validation : Processed queries are validated to prevent injection attacks

Optimization : Query plans may be cached for frequently-executed queries

9.7.3 Multi-Datasource Support

The system supports heterogeneous datasources:

Relational Databases : SQL queries with full `WBPL` placeholder support

`REST` APIs : HTTP requests with parameter substitution and response transformation

NoSQL Databases : Native query languages (MongoDB, Elasticsearch, etc.)

File Systems : Direct file access and processing

9.7.4 Error Handling

Datasource errors are gracefully handled:

- Connection failures trigger automatic retry logic
- Query errors are logged and appropriate error responses are returned
- Partial failures in multi-content pages don't affect other content elements

9.8 Performance Optimization and Timeout Management

The Web Spinner implements several performance and reliability features:

9.8.1 Content Timeouts

Each content element has configurable timeout settings:

Query Timeout : Maximum time allowed for datasource queries

Layout Timeout : Maximum time for layout compilation and rendering

Total Request Timeout : Overall timeout for content delivery

9.8.2 Caching Mechanisms

Multiple levels of caching improve performance:

Query Result Caching : Datasource results are cached based on query signatures
Layout Caching : Compiled layouts are cached until templates change
Response Caching : Complete HTTP responses may be cached for static content

9.8.3 Load Balancing

Multiple Web Spinner instances can operate in parallel:

- Session affinity ensures consistent user experience
- Shared cache layers enable scaling across multiple servers
- Health monitoring ensures failed instances are automatically excluded

9.8.4 Monitoring and Metrics

The Web Spinner provides comprehensive monitoring:

- Request/response times and throughput metrics
- Datasource performance and error rates
- User session analytics and behavior patterns
- System resource utilization and capacity planning data

This architecture ensures that the Web Spinner can handle enterprise-scale workloads while maintaining high performance, security, and reliability standards.

9.9 Looking Forward

At this point, we have a powerful engine and a sophisticated set of languages. However, asking developers to write raw `WBDL` in a text editor, manage resources via the command line, and test queries in a separate database tool would be laborious and inefficient. How can we streamline this development process?

This is where the **Spin the Web Studio** comes in. The Studio, itself a *webbaselet*, provides a comprehensive, integrated development environment for building, testing, and deploying *webbase*s. It is the testing ground for the entire Web Spinner ecosystem and an essential tool for any serious developer. The next chapter will be dedicated to exploring the Studio's rich feature set.

Chapter 10

Spin the Web Studio: An Integrated Development Environment

"The best way to predict the future is to invent it."
— Alan Kay

10.1 The Need for an Integrated Environment

While the `WBDL` and `WBPL` languages provide a powerful declarative framework for defining enterprise portals, and the *Web Spinner* engine offers a robust runtime, the development experience can still be laborious. Without a dedicated toolset, a developer would be forced to:

- Write and edit complex `WBDL` files in a standard text editor, without syntax highlighting, validation, or autocompletion.
- Manage portal resources, such as datasources and user roles, through command-line interfaces or direct database manipulation.
- Create and test `WBPL` queries in a separate database management system, disconnected from the portal context.
- Manually compile and deploy the *webbase* to see the results of any changes.

This fragmented workflow is inefficient, error-prone, and a significant barrier to productivity. To address these challenges, the **Spin the Web Studio** was created.

10.2 Introducing the Spin the Web Studio

The Spin the Web Studio is a web-based development environment designed specifically for building, testing, and managing *webbase*s. It streamlines the entire development lifecycle, from initial design to final deployment, providing a single, coherent interface for all development tasks.

Crucially, the Studio is not an external, standalone application. It is itself a ***webbaselet***, built using the very same technologies it helps to create. This has two profound implications:

1. **The Ultimate Testing Ground**: The Studio serves as the primary testing ground for the *Web Spinner* engine and the entire Spin the Web framework. Every feature of the framework must be robust and performant enough to run the Studio itself.

2. **Part of the Virtualized Portal**: As a *webbaselet*, the Studio can be seamlessly integrated into any *webbase*. This allows developers with the appropriate permissions to access the development environment directly from within the live portal, making it a true part of the virtualized web portal.

10.3 Studio Architecture and Activation

The Studio operates as a specialized *webbaselet* that is added to the portal *webbase* and is accessible to users with developer permissions when they log in. The studio remains hidden until activated by the developer using the keyboard shortcut **Alt+F12**.

Upon activation, the Studio transforms the portal interface into a full-featured development environment with the following key components:

Action Bar Provides quick access to the Studio functions.

Side Bar Houses multiple views including interactive *webbase* hierarchy, portal folder contents, search capabilities, debugging tools, source control integration, and Studio settings.

Status Bar Displays development status information, compilation results, and system feedback.

Panel Contains additional development tools, terminal access, and debugging output.

Main Section Features a tabbed interface where the live portal is displayed in a persistent browser tab alongside other editors.

10.3.1 Unique Browser Tab Integration

One of the Studio's most innovative features is its tab management system. The main section displays the live portal in a special browser tab that cannot be closed, ensuring developers always maintain visual connection to their running application. Portal resources such as files, configurations, and logs can be opened in separate tabs for editing and review.

This design creates an in-place development experience where:

- Changes to `WBDL` files are immediately reflected in the persistent portal view
- Developers can interact with the live portal while simultaneously editing its underlying code
- The development environment becomes part of the portal ecosystem rather than an external tool

10.3.2 Side Bar Navigation

The Studio's side bar provides comprehensive navigation and development tools through multiple specialized views:

Interactive *webbase* Hierarchy A tree view displaying the complete structure of the *webbase*, allowing developers to navigate between *webbaselets*, datasources, and configurations with visual context.

Portal Primary Folder Contents Direct access to the portal's file system, enabling quick file management and resource organization.

Search Comprehensive search capabilities across both *webbase* definitions and folder contents, supporting pattern matching and content searches.

Debugging Integrated debugging tools for `WBPL` queries, *webbaselet* execution, and runtime diagnostics.

Source Control Built-in version control integration for tracking changes to *webbase* definitions and portal resources.

Studio Settings Customization options for the development environment, including themes, editor preferences, and workflow configurations.

10.4 Key Features of the Studio

The Studio is organized into several key modules, each addressing a specific aspect of *webbase* development.

10.4.1 The WBDL Editor

A rich text editor with full support for the `WBDL` syntax, featuring:

- Syntax highlighting for improved readability.
- Real-time validation to catch errors as you type.
- Autocompletion for element names and properties.
- A hierarchical tree view for easy navigation of the *webbase* structure.

10.4.2 The WBPL Query Builder

An interactive tool for creating, testing, and debugging `WBPL` queries:

- A graphical interface for building complex queries.
- The ability to execute queries against live datasources and view the results instantly.
- A "persona simulator" to test queries under different user roles and contexts.

10.4.3 Resource Management

A centralized dashboard for managing all portal resources:

- **Datasource Configuration**: Connect to and manage various data sources (databases, APIs, etc.).
- **User and Role Management**: Define user roles and assign permissions.
- **Asset Library**: Upload and manage static assets like images, CSS, and JavaScript files.

10.4.4 Live Preview and Deployment

The Studio provides a real-time preview of the portal as you build it. Developers can instantly see how their changes will look and behave. When development is complete, the Studio offers one-click deployment to staging or production environments.

10.5 Looking Forward

The Spin the Web Studio completes the conceptual picture of the framework, providing the "workshop" necessary to build with the "blueprint" and the "engine." It transforms *webbase* development from a manual, error-prone task into a streamlined, interactive experience.

Now that we have covered the complete set of tools in the platform—from the declarative languages to the runtime engine and the IDE—the next chapter will ground these concepts in reality. We will examine the specific technology stack and implementation details of the reference Web Spinner, revealing how these architectural principles are translated into running code.

Chapter 11

Technology Stack and Implementation

This chapter documents the concrete technologies used to implement the web spinner mechanics and how the theoretical constructs from Part II are realized in a working system.

11.1 Runtime and Languages

The reference implementation is written in TypeScript and runs on the Deno runtime:

- Deno runtime and TypeScript for server-side logic
- Standard Web APIs (URL, URLSearchParams, Fetch, Crypto) leveraged directly in server code
- No Node.js dependency; tasks and scripts are run via Deno (e.g., CSS/JS minification)
- Containerization via a multi-stage Dockerfile based on denoland/deno:alpine

11.1.1 Why Deno for the *Web Spinner*?

Choosing Deno aligns the engine with the platform's web-first design and keeps the implementation close to the browser runtime:

- **TypeScript end-to-end:** one language for the spinner, utilities, and any shared logic with client code.
- **Web APIs in the server:** standard `URL`, `Request/Response`, `Fetch`, and `Crypto` lower cognitive load and reduce bespoke abstractions.
- **Secure-by-default:** explicit permissions (fs, net, env) encourage disciplined deployment.
- **Batteries included:** built-in formatter, test runner, task runner, and linting simplify tooling.
- **Simple deploys:** reproducible Docker images and a single binary runtime keep operations light.
- **ESM-first:** native modules and top-level `await` fit the compilation model used by `WBLL` and `WBPL` tooling.

Alternatives such as Node.js or Python were considered. Node.js provides a larger ecosystem, but Deno's standard Web API surface and permission model matched the *Web Spinner* 's goals better and reduced dependencies.

11.1.2 What about Python?

Python is a viable alternative for implementing the spinner. A reference implementation could be built with FastAPI or Starlette for HTTP, Jinja2 for small templating needs (separate from `WBLL`), and Pydantic for schemas. Trade-offs:

- **Pros:** mature ecosystem, superb async frameworks (ASGI), rich libraries for parsing, i18n, and data access; strong typing with Pydantic and type hints.
- **Cons:** diverges from TypeScript used elsewhere; requires re-implementing the `WBLL` tokenizer and `WBPL` evaluator; less direct parity with browser APIs.

Key components in a Python port

- **Core model:** classes for `STWSite`, `STWArea`, `STWPage`, `STWContent` with localization and visibility semantics.
- **`WBLL` engine:** tokenizer, validator, and a renderer that emits HTML fragments; map tokens to handlers (inputs, links, lists, media, actions).
- **`WBPL` evaluator:** safe expression parser with interpolation, conditionals, and function library; strict escaping rules.
- **Session and security:** role-based visibility, locale resolution, placeholder hydration; CSRF and auth integration.
- **Data adapters:** pluggable sources (SQL/NoSQL/HTTP) with uniform cursor/record interfaces.
- **Streaming/partials:** endpoints returning fragments and optional batched REST calls; optional WebSocket fan-out.

Porting checklist

1. Define domain classes and serialization compatible with `WBDL`.
2. Implement `WBLL` tokenization, validation, and rendering with test coverage.
3. Implement `WBPL` expression parsing/evaluation with escaping tests.
4. Build the request pipeline: session, lookup, data fetch, placeholder merge, render.
5. Add role/i18n resolution and visibility propagation.
6. Provide fragment endpoints, caching, and optional WebSocket orchestration.
7. Package and deploy (ASGI server, container image, health checks).

11.2 High-Level Architecture

The spinner is a server that understands WBDL. On each request it:[1]

1. Establishes or resumes a session (user, roles, locale, placeholders)
2. Ensures the requested WBDL/Webbase is loaded into an in-memory tree
3. Decides whether to respond with a resource directly or with a list of REST calls the client should make (async via WebSockets)

[1] See also the “Paradigm” section in the implementation README

4. Renders contents on demand using WBLL-driven layouts and returns HTML fragments/resources

11.3 Core Elements and Site Tree

The in-memory model mirrors the WBDL structure:

- **STWSite** (singleton root), **STWArea**, **STWPage**, **STWContent**
- All derive from an abstract **STWElement** providing identity, naming, localization, hierarchy, and export to WBDL
- A content type (e.g., Text, Table, Menus, Breadcrumbs, Calendar, Code Editor, ...), implemented under `stwContents/`, encapsulates data access and rendering concerns

11.4 Rendering Pipeline: WBLL and WBPL

Presentation is described with WBLL (Webbase Layout Language), interpreted by a layout engine:

- WBLL strings are tokenized and validated; tokens drive generation of a specialized render function
- Token handlers cover inputs, lists, links, media, buttons/actions, and structural fragments; they build HTML using placeholders and field values
- WBPL expressions provide string interpolation and conditional/functional logic within layouts and settings
- Placeholders (e.g., `@@name`) merge session, request, and record values

11.5 Request Flow in Practice

For a content render request:

1. The session determines visibility (role-based) and language
2. The content locates its WBLL layout for the current language
3. Records are fetched (via a datasource or parameters); the first row and fields hydrate placeholders
4. The compiled WBLL render function executes, producing the HTML body; optional header/-footer wrappers apply

11.6 Security, Localization, and State

- **Visibility**: role-based flags inherited along the element tree control exposure of nodes
- **Localization**: localized properties (names, slugs, messages) are resolved through the session
- **State**: per-session placeholders and content-level settings influence rendering and actions

11.7 Build, Tooling, and Deployment

- Deno tasks: merge and minify static assets (e.g., CSS merger, JS minifier) for the `public/` client assets
- Tests: parsing and evaluation tests for WBPL ensure correctness of expressions and escaping
- Docker: multi-stage build caches Deno dependencies and ships a non-root runtime image

11.8 Where the Mechanics Live (Guide to Source)

The following folders in the reference implementation contain the mechanics described above:

- `stwElements/`: *STWElement*, *STWSite*, *STWArea*, *STWPage*, *STWContent*
- `stwContents/`: concrete contents and *WBLL* engine (layout parsing/rendering)
- `tests/`: WBPL and layout-related unit tests
- `public/`: client-side scripts, styles, and SPA shell
- `tasks/`: Deno-powered dev/build utilities (e.g., minification, CSS merge)

11.9 Example: From WBDL to HTML

At a glance:

1. WBDL defines the site tree; the spinner builds an in-memory model at startup/load
2. A user navigates to a page; the spinner locates the route and the associated contents
3. Each content loads data (if needed), prepares placeholders, and renders its WBLL layout
4. The server responds with an HTML fragment or instructs the client to fetch multiple fragments via REST/WebSockets

This chapter has bridged theory and implementation, showing how the abstract mechanics of the Web Spinner are realized in a modern technology stack. With the platform's construction now fully detailed—from its languages to its engine and development tools—our focus shifts from building the framework to using it.

The next part of this book begins the practical guide for developers, covering the development models and methodologies for designing and building effective web portals with the platform we have just described.

Part III

The Web Portal

Introduction to Part III: The Web Portal

"Structure is not just a means to a solution. It is the solution."
— Alexandra V. Agranovsky

With the Spin the Web framework fully specified in Part II, this part transitions from engineering blueprints to practical application. It serves as the developer's guide to the models and methodologies for *using* the framework to design, build, and structure a real-world enterprise portal.

This part covers the methodology for translating business requirements into a logical portal structure and explores the learning path and mindset required to become a proficient portal developer. Throughout, we reference the public website spintheweb.org and the companion *Spin the Web Studio*—both built with the framework—as running examples of the patterns described here. For an overview of the community catalog used to discover and rate webbaselets, see the Ecosystem webbaselet in appendix B.3.

Chapter 12: Structuring a Web Portal (chapter 12) – A practical guide to designing a portal's structure based on business needs and user journeys.

Chapter 13: The Portal Development Journey (chapter 13) – Explores the patterns, nomenclature, and mindset required for effective portal development.

By the end of this part, you will have a clear methodology for designing and building sophisticated web portals using the Spin the Web framework, with spintheweb.org and Spin the Web Studio serving as concrete, navigable references that showcase information architecture, webbaselets, workflows, and search in practice.

Chapter 12

Structuring a Web Portal: A Practical Guide

"Structure is not just a means to a solution. It is the solution."
— Alexandra V. Agranovsky

12.1 Step 1: Map Business Functions to Areas

A company's partial or total structure can be projected in a web portal: the lobby, human resources, sales, marketing, production, logistics, administration, service, support... you name it. Each of these areas serves functionalities that address specific needs. Some functionalities satisfy cross-company needs, some are public while others are private, and some expose everything while others only a part based on authorization.

The preceding chapters have detailed the languages and mechanics of the Spin the Web framework. This chapter provides a practical guide on how to translate an organization's business structure and user needs into a coherent and effective web portal using the core `WBDL` elements: `STWArea`, `STWPage`, and `STWContent`.

The key to a successful portal is a structure that feels intuitive to its users. This is achieved by mirroring the familiar functions of the business itself. We apply a three-step approach: (1) map business functions to Areas, (2) design Pages for user journeys, and (3) build Pages with Content blocks.

12.1.1 Mapping Business Functions to Areas

The first step is to identify the primary functional divisions of the organization. These will become the top-level **Areas (`STWArea`)** of your portal. Think of these as the main departments or directorates of the company. The organization's official organizational chart is often a practical starting point for this mapping, providing a clear, agreed structure that can be refined for digital navigation needs.

A typical manufacturing company might have the following top-level Areas:

- **Sales**: For customers, sales representatives, and channel partners.
- **Administration**: For finance, HR, and internal services.

- **Backoffice**: For logistics, procurement, and supplier management.
- **Technical Office**: For engineering, R&D, and product support.
- **Products & Services**: A publicly accessible area showcasing the company's offerings.

Each of these would be defined as an `STWArea` element in your `WBDL` file, forming the main navigation structure of your portal.

12.1.2 Documentation at the Area Level

Each Area definition should include documentation that serves both operational and compliance purposes. For example, the Sales Area might include:

```
{
  "_id": "sales",
  "type": "Area",
  "name": {
    "en": "Sales"
  },
  "description": {
    "en": "Sales Department Operations Center. This area supports the complete sales
    ↪ lifecycle from lead generation to order fulfillment. All activities comply with our
    ↪ Customer Relationship Management Policy CRM-POL-001. Key Processes: Lead
    ↪ qualification (PROC-SALES-001), Quote generation (PROC-SALES-002), Order processing
    ↪ (PROC-SALES-003). Performance Targets: Response time to quotes: <24 hours, Customer
    ↪ satisfaction: >90%. Department Manager: Jane Smith (ext. 2001)."
  },
  "keywords": {
    "en": "sales, CRM, leads, quotes, orders, customer service, KPI"
  }
}
```

Listing 12.1: Sales Area with Documentation

This approach ensures that organizational knowledge, procedures, and quality standards are embedded directly within the portal structure, creating a living documentation system that evolves with the business.

12.2 Documentation Through Structure

Before diving into the structural methodology, it's crucial to understand that every `WBDL` element—from the root `STWSite` down to individual `STWContent` blocks—includes localized `keywords` and `description` elements. These serve far more than basic metadata; they provide the infrastructure for embedding organizational knowledge directly within the portal structure.

12.2.1 Quality Management Integration

This hierarchical documentation system transforms the portal into a comprehensive organizational knowledge base where quality management principles, procedures, and manuals are seamlessly integrated with the functional structure. Each level serves specific documentation purposes:

Site Level : Overall company quality policy, mission statement, and enterprise-wide standards

Area Level : Department-specific procedures, compliance requirements, and operational standards

Page Level : Specific process documentation, workflows, and user guidelines

Content Level : Individual field instructions, data validation rules, and contextual help

12.2.2 Living Documentation Benefits

This approach provides several critical advantages for enterprise implementation:

- **Living Documentation**: Procedures stay current with actual portal functionality, eliminating outdated manuals
- **Contextual Access**: Users access relevant quality information exactly where they need it in their workflow
- **Compliance Tracking**: Keywords enable systematic auditing and compliance reporting across the organization
- **Multi-Language Support**: Quality documentation can be localized for global operations
- **Search Integration**: The *Spin the Web Studio* search capabilities can locate quality procedures across the entire portal structure
- **Audit Trails**: Every element becomes a potential compliance checkpoint with embedded documentation

Consider this example of an Area with embedded quality management documentation:

```
{
  "_id": "qa_control",
  "type": "Area",
  "name": {
    "en": "Quality Control"
  },
  "description": {
    "en": "This area implements ISO 9001:2015 Section 8.5 - Production and Service
    ↪ Provision. All processes documented here follow our Quality Manual QM-2024-Rev3.
    ↪ Key Performance Indicators: Defect Rate < 0.1%, Customer Satisfaction > 95%."
  },
  "keywords": {
    "en": "quality, ISO 9001, production, service, KPI, audit"
  }
}
```

Listing 12.2: Area with Quality Management Documentation

This embedded documentation makes the portal self-describing, turning it into a living quality management system.

12.3 Core Page Archetypes and Navigation

Structure vs. Branding This chapter focuses on structural semantics: `STWArea`, `STWPage`, `STWContent`, and page archetypes. Visual identity is brand-specific and orthogonal to structure—whatever the styling, a table remains a table and a form remains a form. Keep navigation and information architecture stable; apply branding through themes, CSS, components, and iconography without changing the underlying structure.

Navigation is task-driven and often crosses Area boundaries. A representative flow is: start from a customers list, search/select a customer, view their orders, open a specific order, jump to a purchased item, and inspect its manufacturing history, commissioning, maintenance, spare parts, and last sales prices. These flows should be implemented as explicit links and contextual actions between `STWPages` so users can traverse the business graph naturally.

Navigation is provided through **main menus**, **side menus**, **breadcrumbs**, **inline hyperlinks**, **interactive maps**, and **iconographic menus** (cards/tiles). Use maps when location is a primary dimension; use iconography for rapid recognition and **mobile-friendly navigation**.

12.3.1 Three Fundamental Main Content Archetypes

Regardless of domain, the main body of most pages is built from three fundamental archetypes:

- **Dashboards** — summaries, KPIs, and shortcuts for quick situational awareness.
- **Tabular Data** — searchable/filterable tables or lists for browsing and selection; can be regular tables, drillable pivot tables, or geomaps when data is spatial.
- **Details** — focused views for inspecting or editing a single entity.

12.4 Step 2: Design Pages for User Journeys

Once Areas are defined, the next step is to design the **Pages (`STWPage`)** that will live within them. Each Page should correspond to a specific user task or "journey." For example, within the Sales Area, you might have Pages for "New Lead Entry," "Quote Management," and "Customer Dashboard."

Pages are divided into sections: header, sidebars, main body, and footer. Each of these sections may hold zero or more contents. For example, the header provides transversal navigation, search, assistance, ticketing, profile management, and cultural preferences. The sidebars offer specific navigation, filtering, legends, attachments, info, and summaries. The main body can contain anything from dashboards to tables, forms, lists, and calendars. The footer typically includes contacts and feedback options.

This modular structure allows for consistent yet flexible page layouts across the portal.

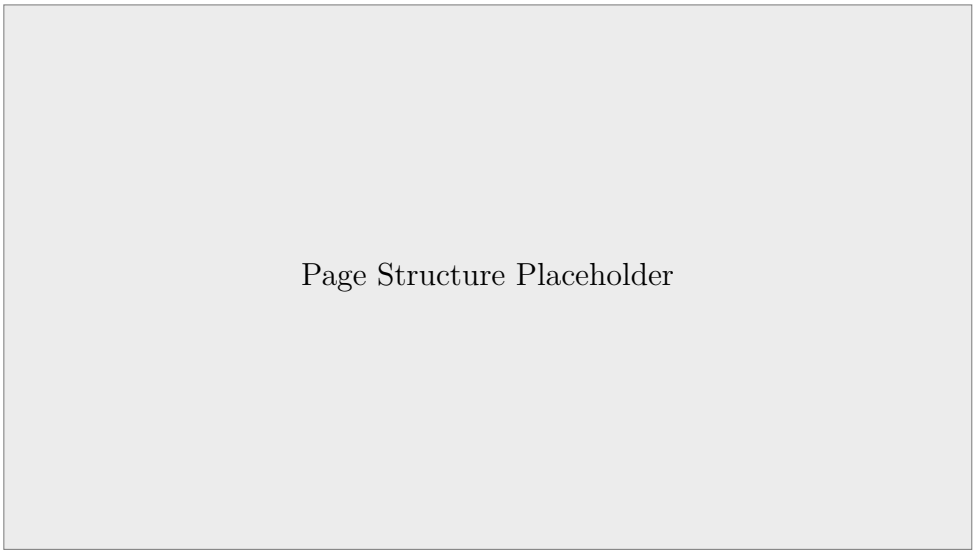


Figure 12.1: Standard Page Structure

12.4.1 Documentation at the Page Level

Pages should document specific workflows and processes. For example, the "Quote Generator" page might include detailed process documentation:

```
{
  "_id": "quote_generation_page",
  "type": "Page",
  "name": {
    "en": "Generate New Quote"
  },
  "description": {
    "en": "This page implements process PROC-SALES-002 for generating new customer quotes.
    ↪ The process requires completion of the customer information form and selection of
    ↪ products from the official catalog. All generated quotes must be approved by a
    ↪ sales manager before being sent to the customer. This process is audited quarterly
    ↪ under audit procedure AUD-Q-004."
  },
  "keywords": {
    "en": "quote, sales process, PROC-SALES-002, audit, compliance"
  }
}
```

Listing 12.3: Page with Process Documentation

12.4.2 Integrated Ticketing and Feedback

A professional portal must include a **ticketing system** that captures and tracks requests beyond products or services. Tickets should also address the *portal structure and behavior* itself (navigation, layout, performance, accessibility, localization), as well as data quality, access/permissions, and feature requests.

Recommended capabilities:

- **Entry points**: a global action in the header (Help/Support), and an *in-context* action near the main content (e.g., "Report an issue with this page").
- **Automatic context**: include URL, Area/Page/Content identifiers, user locale, device, and current filters/scope when applicable.
- **Classification**: category (product/service, portal structure, portal behavior/UX, data issue, access/permissions, feature request), severity/impact, attachments, and tags.
- **Workflow**: triage, assignment, SLA targets, status, and audit trail. Integrate with quality management and release notes.
- **Discoverability**: use localized `keywords` and `description` to improve internal SEO for ticket templates and knowledge-base links.

Example of a ticket payload embedding structural context:

```
{
  "_id": "tick-2025-09-09-1234",
  "type": "Ticket",
  "category": "portal_structure",
  "summary": { "en": "Sidebar filters hidden at 1280px" },
  "description": { "en": "On Customers page, the left filter panel collapses unexpectedly
    ↪ at 1280px width." },
  "scope": {
    "site": "stw_site",
    "area": "sales",
    "page": "customers_list",
    "content": "customers_table"
  },
  "context": {
    "url": "/sales/customers",
    "locale": "en",
    "viewport": "1280x720",
    "filters": { "status": "active" }
  },
  "severity": "medium",
  "status": "new",
  "attachments": [],
  "keywords": { "en": "ux, layout, sidebar, filters" }
}
```

Listing 12.4: Ticket with Structural Context

This approach ensures issues are actionable and traceable, while keeping the feedback loop close to where users work.

12.5 Step 3: Build Pages with Content Blocks

The final step is to populate the pages with **Content (`STWContent`)** blocks. These are the building blocks that display information and provide functionality. A single page will typically be composed of multiple content blocks.

Let's design the public-facing **"Products"** page from the "Products & Services" Area. This page needs to be informative and engaging for potential customers. It could be built with the following `STWContent` blocks:

1. **A Hero Banner**: An eye-catching banner image with a marketing headline. This is a static content block.
2. **Product Categories Menu**: A navigation menu, perhaps implemented as tabs or an accordion, that allows users to filter products by category. This content block would use `WBPL` to dynamically query the list of product categories from a database.
3. **Product Listing Grid**: A grid that displays the products. Each item in the grid would show a product image, name, and a short description. This is a highly dynamic content block, driven by a `WBPL` query that fetches the product data based on the selected category.
4. **Featured Products Carousel**: A rotating carousel showcasing new or featured products. This could be driven by a separate, specialized query.
5. **Call to Action**: A block encouraging users to "Request a Quote" or "Contact Sales," linking to another page in the portal.

By combining these static and dynamic content blocks, you can create a rich, interactive, and data-driven page that effectively serves the needs of its audience.

12.5.1 Documentation at the Content Level

Individual content blocks can contain specific field-level instructions and validation rules. For example, a customer information form might include:

```
{
  "_id": "customer_info_form",
  "type": "Content",
  "subtype": "Form",
  "name": {
    "en": "Customer Information"
  },
  "description": {
    "en": "Customer Data Collection Form - FORM-CRM-001. This form collects essential
    ↪ customer information required for quote generation. All fields marked with (*) are
    ↪ mandatory as per our Customer Data Policy CDP-001. Data Validation Rules: Company
    ↪ Name (min 2, max 100 chars), Email (valid business format). Data Protection Notice:
    ↪ Customer data is processed according to GDPR Article 6(1)(b). Data retention
    ↪ period: 7 years."
  },
  "keywords": {
    "en": "customer data, GDPR, validation rules, data quality, mandatory fields"
```

```
  }
}
```

Listing 12.5: Content with Field-Level Documentation

This multi-level documentation approach ensures that quality management principles, compliance requirements, and operational procedures are seamlessly integrated throughout the portal structure, creating a comprehensive knowledge management system that supports both daily operations and audit requirements.

12.5.2 Search Modes and SEO

The localized `keywords` and `description` fields on every `WBDL` element are not just documentation—they power internal SEO within the portal. They improve discoverability in menus and sitemap views, drive search ranking and snippets in *Spin the Web Studio*, and help generate meaningful breadcrumbs and suggestions.

There are two complementary search experiences:

Site-wide Search traverses Areas, Pages, and Content using localized metadata (`name`, `keywords`, `description`) and usage signals. It is typically exposed in the page header and returns cross-area results.

In-context Search operates within the active page. It narrows scope to the main body: filtering rows in tables/lists, quick find in forms, text search in rich content, and local help/glossary. Place it near the primary content (e.g., table toolbar or section header) for immediate focus.

Design guidance: keep global search persistent in the header; keep in-context search adjacent to the content it filters. Ensure accessibility (labels, ARIA, keyboard shortcuts like / to focus, Esc to clear) and support localization for both search modes.

12.6 Looking Forward

This chapter has provided a practical methodology for structuring a web portal by mapping business functions to Areas, user journeys to Pages, and informational needs to Content blocks. This "documentation through structure" approach ensures that the resulting portal is both logical and self-describing.

However, building a successful portal requires more than just technical structure; it requires a specific mindset and an understanding of recurring patterns. The next chapter delves into the learning journey of a portal developer, exploring the key patterns and nomenclature that emerge from experience.

Chapter 13

Learning from Experience: The Portal Development Journey

If you focus on a given subject long enough, you'll discover patterns. These patterns give way to insights that lead to new patterns; the subject acquires complexity, during this natural evolution these patterns can be generalized by making small adjustments, often reducing complexity while extending reach. Experience is the force that drives complexity oscillations, at its apex, it gives way to simplicity. And that is beautiful!

This chapter explores the nomenclature and patterns that have emerged through years of web portal development, providing insights into the systematic approach that led to Spin the Web.

13.1 The Evolution of Enterprise Web Presence

The journey from simple websites to comprehensive portal solutions reflects the natural evolution of enterprise digital presence. Understanding this progression is crucial for appreciating the systematic approach embodied in Spin the Web.

13.1.1 Classification by Audience

The terminology surrounding enterprise web presence has evolved significantly since the late 1990s. What began as simple "websites" gradually transformed into more sophisticated platforms based on their target audience:

Table 13.1: Classification of Enterprise Web Platforms by Audience

Audience Type	**Platform Classification**	**Primary Purpose**
Company employees	Intranet	Internal collaboration and data access
Clients and suppliers	Extranet	B2B interactions and transactions
General public	Website	Marketing and public information
Mixed audience	Portal	Unified, role-based access
Stakeholders	Enterprise Portal	Comprehensive organizational interface

This classification system emerged from practical necessity as organizations recognized that different audiences required different levels of access, functionality, and user experience design.

13.2 The Data Fragmentation Challenge

13.2.1 The Reality of Enterprise Data

One of the fundamental challenges that led to the development of portal technologies was the recognition of how disparate and dispersed company data had become. Modern organizations typically manage information through a complex ecosystem of specialized software:

- **Enterprise Resource Planning (ERP)** systems for core business processes
- **Customer Relationship Management (CRM)** platforms for customer interactions
- **Product Data Management (PDM)** systems for product information
- **Warehouse Management Systems (WMS)** for inventory control
- **Project Management Systems (PMS)** for project coordination
- **Business Process Management (BPM)** tools for workflow automation
- Traditional tools: spreadsheets, databases, files, email systems

This fragmentation creates what can only be described as a "data jungle"—a complex, often impenetrable ecosystem where information exists in silos, each with its own interface, data model, and access paradigm.

13.2.2 The Vision of Harmony

The emergence of web technologies in the late 1990s presented an opportunity to transform this jungle into something more harmonious, coherent, and elegant. The key insight was that web technologies could serve as a unifying layer, creating a single interface that could:

1. Reflect company branding and design guidelines.
2. Reflect company organizational structure and roles.
3. Ensure robust security and access controls.
4. Query multiple data sources simultaneously.
5. Present information in a coherent, tailored graphical interface.
6. Enable intuitive data inspection and manipulation.
7. Support complete CRUD (Create, Read, Update, Delete) operations across systems.
8. Provide a seamless user experience across diverse data sources.
9. Enable real-time data updates and notifications.
10. Facilitate collaboration and communication among users including support for messaging, file sharing, and joint editing.
11. Web portal ticketing and issue tracking capabilities.
12. The list goes on...

13.3 The Developer's Perspective

13.3.1 Full Stack Development Philosophy

The portal development approach requires a specific mindset that balances technical capability with user experience design. This perspective focuses on:

Data Management Expertise:

- Data collection strategies
- Information organization principles
- Processing and storage optimization
- Retrieval and presentation techniques
- Data exploration interfaces

User Interface Design:

- Role-based interface customization
- Multifaceted interaction patterns
- Consistency across diverse data sources
- Intuitive navigation structures

13.3.2 The Balance with Marketing

Portal development must balance functional requirements with marketing objectives. While marketing drives the need for engaging, persuasive interfaces that promote products and services, the core portal functionality requires:

- Sobriety in design for daily operational use
- Style consistency across all functional areas
- Clarity in data presentation
- Efficiency in task completion

The successful portal designer appreciates both the flashy, marketing-driven areas and the sober, functionality-focused regions, ensuring that both serve their intended purposes without compromising the overall user experience.

13.4 Pattern Recognition and Generalization

13.4.1 The Natural Evolution of Complexity

Through extended focus on portal development, distinct patterns emerge that follow a predictable cycle:

1. **Initial Pattern Recognition**: Identifying recurring challenges and solutions
2. **Insight Development**: Understanding the underlying principles
3. **New Pattern Formation**: Creating enhanced approaches based on insights
4. **Complexity Accumulation**: Integration of multiple patterns and solutions
5. **Generalization**: Abstraction and simplification through experience
6. **Elegant Simplicity**: The apex of understanding where complex problems have simple solutions

This cycle represents the natural evolution of expertise, where experience becomes the driving force behind complexity oscillations.

13.4.2 Key Patterns in Portal Development

Several fundamental patterns have emerged through years of portal development:

Virtualized Company Pattern: The concept of presenting the entire organization through a single, coherent digital interface that adapts to user roles and needs.

Hierarchical Namespace Pattern: Organizing content and functionality in logical, navigable structures that reflect organizational hierarchies and business processes.

Role-Based Presentation Pattern: Dynamically adjusting interface elements, available functions, and visible data based on user roles and permissions.

Integrated Data Source Pattern: Seamlessly combining information from multiple backend systems into unified presentations.

13.5 The Learning Framework

13.5.1 Core Learning Areas

The systematic study of portal development encompasses several key areas that form the foundation of expertise:

Organizational Structure Understanding how organizational hierarchies translate into digital interfaces and access patterns.

Authorization Systems Designing comprehensive permission models that support complex organizational roles while maintaining security and usability.

Information Architecture Creating logical, scalable structures for organizing and presenting diverse types of content and functionality.

Search and Discovery Implementing sophisticated search capabilities that work across multiple data sources and content types.

Help and Documentation Developing context-sensitive assistance systems that support users at all skill levels.

Data Integration Mastering the technical and conceptual challenges of combining disparate data sources into coherent presentations.

Presentation Layer Design Creating flexible, adaptable interfaces that can accommodate diverse content types and user needs.

Interaction Design Developing intuitive, efficient interaction patterns that support complex workflows while remaining accessible.

13.5.2 The Nomenclature of Portal Development

Professional portal development has developed its own vocabulary that reflects the unique challenges and solutions in this field:

- **Webbase**: The complete definition of a portal's structure, content, and behavior
- **Webbaselet**: A modular component that can be integrated into larger portal structures
- **Web Spinner**: The engine that interprets portal definitions and generates user experiences

- **Virtualized Company**: The comprehensive digital representation of an organization
- **Role-Based Presentation**: Dynamic interface adaptation based on user roles
- **Hierarchical Namespace**: The logical organization of portal content and functionality

This specialized vocabulary enables precise communication about complex concepts and facilitates the transfer of knowledge between developers and stakeholders.

13.6 Practical Applications

13.6.1 Learning Through Implementation

The concepts and patterns described in this chapter are best understood through practical application. Each learning area provides opportunities for hands-on exploration:

1. Start with simple organizational structures and gradually increase complexity
2. Implement basic authorization systems before tackling enterprise-scale permission models
3. Practice with small-scale data integration before attempting comprehensive system integration
4. Develop simple interaction patterns before creating complex workflow support

13.6.2 Pattern Application in Real Projects

Real-world portal projects provide the best laboratory for understanding these patterns:

- Employee portals demonstrate organizational structure and authorization patterns
- Customer portals showcase role-based presentation and data integration challenges
- Partner portals illustrate complex permission models and workflow integration
- Public-facing portals emphasize presentation layer design and search capabilities

13.7 The Path to Mastery

13.7.1 Experience as the Driver

The journey from complexity to simplicity is driven by experience—the accumulated understanding that comes from repeated engagement with challenging problems. This experience manifests as:

- Recognition of fundamental patterns beneath surface complexity
- Ability to see elegant solutions to apparently complex problems
- Understanding of the trade-offs inherent in different approaches
- Intuition about what will work in specific contexts

13.7.2 The Beauty of Simplicity

At the apex of experience, complex problems reveal simple solutions. This is the goal of the systematic approach embodied in Spin the Web: to provide a framework that makes the complex simple, the difficult elegant, and the overwhelming manageable.

The beauty lies not in the complexity of the solution, but in its ability to handle complexity with apparent simplicity—creating powerful, flexible portal solutions that feel natural and intuitive to both developers and users.

13.8 Conclusion

The learning journey in portal development is one of continuous pattern recognition, insight development, and simplification. Spin the Web represents the culmination of this journey—a systematic approach that embodies years of experience in a framework that makes sophisticated portal development accessible and elegant.

By understanding the patterns and principles outlined in this chapter, developers can more effectively leverage the Spin the Web framework to create portal solutions that truly serve as virtualized companies—comprehensive, adaptive digital representations that evolve with their organizations and users.

Part IV

The Future

Introduction to Part IV

This part looks ahead: emerging patterns, research directions, and opportunities for the Spin the Web ecosystem. It consolidates lessons from earlier parts and projects them into near- and long-term roadmaps.

Areas of exploration include interoperability with evolving web standards and APIs, performance and cost at scale, formal methods for validating specifications and transformations, and community-driven extension mechanisms.

A living roadmap. This book is a continuous work in progress. Part IV serves as a staging area for forward-looking ideas and experiments. As proposals mature and are implemented, they are promoted to:

++Part II (The Platform) — when they affect the framework's languages, runtime, or tooling.
Part III (The Web Portal) — when they shape methods, patterns, or models for portal design and delivery.

The chapter that follows, *Future Directions* (chapter 14), collects the most promising threads and defines criteria for promotion into the main body of

Chapter 14

Future Directions: Toward Intelligent Digital Ecosystems

This chapter explores the transformative potential of Spin the Web as it evolves beyond individual portal development toward intelligent, interconnected business ecosystems. We examine how AI agents can enhance portal functionality while envisioning a future where structured corporate portals enable seamless machine-to-machine communication across global business networks.

14.1 Agentic UX

- Contextual copilots embedded within areas/pages
- Task-oriented flows that orchestrate multiple contents and APIs
- Natural-language prompts mapped to parameterized actions

14.2 Knowledge and Reasoning

- Retrieval pipelines grounded in the webbase, logs, and domain docs
- Guardrails and policy evaluation layered over actions
- Transparent, auditable traces for enterprise adoption

14.3 Learning Loops

- Implicit feedback from usage, explicit ratings for outcomes
- Continuous improvement of layouts, queries, and flows
- Safe experimentation with feature flags and A/B variants

14.4 Operational Model

- Private inference endpoints and on-prem models for compliance
- Event streams for real-time adaptations and monitoring
- Cost controls, quotas, and quality-of-service tiers

14.5 Standards and Interop

- Schema-first contracts for tools and agent actions
- Portable traces and evaluation datasets
- Alignment with emerging agent frameworks and security best practices

14.6 The Digital Ecosystem Vision: Machine-to-Machine Business Communication

Envision a digital world where every company maintains a structured web portal built on declarative principles—a world where business communication transcends human interfaces to enable seamless machine-to-machine interaction. In this ecosystem, corporate portals would serve as standardized digital representations of organizations, with each portal exposing structured data through consistent `WBDL` definitions and `WBPL` interfaces that machines can interpret and interact with automatically.

Consider the transformative potential: when suppliers, customers, partners, and service providers all maintain portals with standardized structures, business processes could become truly automated. A manufacturing company's portal could automatically query supplier portals for inventory levels, pricing updates, and delivery schedules. Customer portals could seamlessly integrate with vendor systems for real-time order tracking, automated reordering, and predictive maintenance scheduling. Regulatory compliance could be achieved through automated data exchange between corporate portals and government systems.

This vision extends the concept of eBranding into eMachineReading—where organizations design their digital presence not only for human stakeholders but also for automated business processes. The hierarchical documentation principles embedded within `WBDL` structures would provide the semantic foundation necessary for machines to understand business context, process flows, and data relationships. Quality management metadata embedded in portal structures would enable automated verification, compliance checking, and performance monitoring across entire business networks.

The Spin the Web framework's emphasis on declarative languages and embedded documentation makes this vision achievable. When every business decision, process, and data element is described through structured `keywords` and `description` attributes, the resulting portals become self-documenting APIs that both humans and machines can navigate with equal effectiveness. This represents a paradigm shift from today's fragmented digital business landscape toward an interconnected ecosystem where organizational boundaries become permeable to automated business intelligence while maintaining security and appropriate access controls.

Such a digital ecosystem would fundamentally reshape how businesses discover, evaluate, and engage with one another, creating unprecedented opportunities for efficiency, innovation, and global economic integration through structured digital representation.

14.7 Closing Thoughts

Intelligent agents augment the portal rather than replace it. WBLL and WBDL remain the bedrock, with agents acting as power users that can read, write, and reason across the webbase to accelerate meaningful work.

Appendix A

WBLL Token Reference

This appendix provides a complete reference for WBLL tokens.

General syntax: `<token>('arg1[;arg2][;arg3]...')`

Brackets denote optional arguments:

- No arguments: `e`
- One argument: `e('arg1')`
- Two arguments: `e('arg1;arg2')`
- Three arguments: `e('arg1;arg2;arg3')`

If you need to skip a middle argument, use an empty segment: e.g., `e(';username')` omits *format*.

Formatting note: the *format* argument applies only to numeric output in `e` and `f` (e.g., `€ #,##0.00`, `#,##0`, `0`). For non-numeric values it is ignored; inputs default to `text`.

Cursor movement policy: **editing tokens** (form controls such as `h`, `e`, `w`, `m`) advance the field cursor when the *name* is not specified; **non-editing tokens** advance only when they implicitly consume the active field *value*.

Example recordset (fields in order): `price`, `username`, `city`.

price	username	city
1234.5	alice	Milan
99	bob	Paris
0	carol	Berlin

Assumptions for examples: unless stated otherwise, (1) examples evaluate the first row; (2) the field cursor starts at the first field (`price`); (3) when currency output like "€ 1,234.50" appears, it is the formatted `price` of row 1 using the pattern `€ #,##0.00`.

Token	`/* comment text */`
Description	Multiline comment. The interpreter ignores everything between `/*` and `*/`. Can span multiple lines. Produces no output and does **not** move the field cursor.
Example	Code and output below.

```
lf /* comment
   spanning
   lines */
\rl('UserName')f
```

Renders:

```
<label>price</label>1234.5<br><label>UserName</label>alice
```

Token	`// comment text`
Description	Line comment. The interpreter ignores everything from `//` to the end of the line. Produces no output and does **not** move the field cursor.
Example	Code and output below.

```
t('Hello') // inline comment after code
\r
// full-line comment
t('World')
```

Renders:

```
Hello<br>World
```

Token	`<`
Description	Moves the field cursor back by one position. Produces no output. If the cursor is already at the first field, this operation is a no-op.
Example	Code and output below.

```
>>><f
```

Renders:

```
Milan
```

Token	`>`
Description	Advances the field cursor by one position. Produces no output. If the cursor is already at the last field, this operation is a no-op.
Example	Code and output below.

```
>f
```

Renders:

```
alice
```

Token	`\A('attr="value" ...')`
Description	Like **\a**, but applies attributes to the *parent* structural element: the nearest enclosing `<tr>` (in table contexts) or `<li>` (in list contexts). Non-editing; produces no output and does **not** move the field cursor. Has effect only when inside a table row or list item context. The attributes are processed by the WBPL interpreter.
Example	Code and output below.

```
// In a table row context (created by the table subtype)
\A('class="highlight"')e

// In a list item context (created by the list/menu subtype)
\A('data-role="user"')>e(';username')
```

Renders:

```
<tr class="highlight">... <input type="text" name="price" value="1234.5"> ...</tr>
<li data-role="user">... <input type="text" name="username" value="alice"> ...</li>
```

Token	`\a('attr="value" ...')`
Description	Assigns HTML attributes to the most recently emitted element. Use immediately after a token that creates an element (e.g., `e`, `w`, `m`, `h`). Attributes are appended/merged; later attributes override earlier ones when duplicated. Non-editing; produces no output and does **not** move the field cursor. The attributes are processed by the WBPL interpreter.
Example	Code and output below.

```
e\a('required style="color:red"')
>e(';username')\a('maxlength="20"')
```

Renders:

```
<input type="text" name="price" value="1234.5" required style="color:red"><input
    ↪ type="text" name="username" value="alice" maxlength="20">
```

Token	`\r`
Description	Inserts a line break: ` `. Non-editing; produces no value output and does **not** advance the field cursor.
Example	Code and output below.

```
f\rf
```

Renders:

```
1234.5<br>alice
```

Token	`e('format[;name][;value]')`
Description	Renders a input element. Arguments are *positional but optional*. Defaults: *format* = identity, *name* = active field name, *value* = active field value. **Cursor rule (editing):** the cursor advances when *name* is not specified; if *name* is provided, the cursor does not advance (even if *value* is omitted). **Formatting:** *format* applies only to numeric values.
Example	Code and output below.

```
e('€ #,##0.00;price;1234.5') // All args explicit: no cursor advance; numeric format
    ↪ applied
e // Use active field name and value: advances
e(';username') // Explicit name; uses active field value (username); no cursor
    ↪ advance
```

Renders:

```
<input type="number" name="price" value="€ 1,234.50"><input type="number"
    ↪ name="price" value="1234.5"><input type="text" name="city" value="Milan">
```

Token	`f('format')`
Description	Renders the active field value at the cursor, optionally applying a numeric *format*. The *format* argument is positional but optional (default: identity). **Cursor rule (non-editing):** advances because it consumes the active field value. **Formatting:** applies only to numeric values.
Example	Code and output below.

```
f('€ #,##0.00')\r // Currency formatting of active numeric field
f // No format (identity)
```

Renders:

```
€ 1,234.50<br>alice
```

Token	`h('name[;value]')`
Description	Inserts an HTML input element of type hidden. Arguments are *positional but optional*. Defaults when omitted: *name* = active field name, *value* = active field value. **Cursor rule (editing):** the cursor advances when *name* is not specified (e.g., just **h**); if *name* is provided, the cursor does not advance.
Example	Code and output below.

```
h
h('kind;area')
h('id')
```

Renders:

```
<input type="hidden" name="price" value="1234.5"><input type="hidden" name="kind"
    ↪ value="area"><input type="hidden" name="id" value="alice">
```

Token	`j('code')`
Description	Injects a script element. The WBPL engine processes *code* first (placeholders, expressions), then WBLL emits `<script>`code`</script>`. Non-editing; does **not** move the field cursor.
Example	Code and output below.

```
j('console.log("Rows:", @@rows)')
```

Renders:

```
<script>console.log("Rows:", 3)</script>
```

Token	`k('name[;value]')`
Description	Sets a session variable. Arguments are *positional but optional*. Defaults: *name* = active field name, *value* = active field value. Produces no output. **Cursor rule (non-editing):** advances only when it implicitly uses the active field *value* (i.e., when *value* is omitted). If *value* is provided, the cursor does not advance.
Example	Code and output below.

```
k('role;admin')
k
```

Renders:

Token	`l('label')`
Description	Renders a label for the active field's *name*. If a string argument is provided as `'label'`, it overrides the current field name. Produces no field value output and **does not** advance the cursor.
Example	Code and output below.

```
lf
\rl('User Name')f
```

Renders:

```
<label>price</label>1234.5<br><label>User Name</label>alice
```

Token	`m('name[;value]')`
Description	Renders a multiline text area: `<textarea name="<name>"><value></textarea>`. Arguments are *positional but optional*. Defaults: *name* = active field name, *value* = active field value. **Cursor rule (editing):** advances when *name* is not specified; if *name* is provided, does not advance.
Example	Code and output below.

```
m('comments;Hello, world!')m>m(';New York')
```

Renders:

```
<textarea name="comments">Hello, world!</textarea><textarea
    ↪ name="price">1234.5</textarea><textarea name="city">New York</textarea>
```

Token	`\s('attr="value" ...')`
Description	Sets content-level attributes: `caption`, `header`, `footer`, `key`, `visible`, `enabled`, `disabled`, `nodata`. Non-editing; does not move the field cursor. Multiple calls accumulate; later values override earlier ones for the same key.
Example	Code and output below.

```
\s('caption="Users" header="Found: @@rows" key="fId"') l f l f l f
```

Renders (table subtype):

```
<table data-key="fId">
  <caption>Users</caption>
  <thead>Found: 3</thead>
  <tbody>
    <tr>
      <th>price</th><td>1234.5</td>
      <th>username</th><td>alice</td>
      <th>city</th><td>Milan</td>
    </tr>
    ...
  </tbody>
</table>
```

Token	`t('text')`
Description	Inserts the given text string into the response flow at the current position. Non-editing; does not move the field cursor.
Example	Code and output below.

```
t('Hello World!')
```

Renders:

```
Hello World!
```

Token	`w('name[;value]')`
Description	Renders a password input element: `<input type="password" name="<name>" value="<value>">`. Arguments are *positional but optional*. Defaults: *name* = active field name, *value* = active field value. **Cursor rule (editing):** advances when *name* is not specified; if *name* is provided, does not advance. *Note:* some browsers ignore preset password values.
Example	Code and output below.

```
w('pwd;secret')\rw\rw(';password')
```

Renders:

```
<input type="password" name="pwd" value="secret"><br><input type="password"
    ↪ name="price" value="1234.5"><br><input type="password" name="username"
    ↪ value="password">
```

Appendix B

Webbaselets: BPMS, PLM, and Ticketing

This appendix summarizes three foundational webbaselets that commonly ship with a Spin the Web deployment. Each is delivered as a standalone webbaselet (root STWArea) that can be imported into any Webbase, sharing common conventions for navigation, security, and data.

- Business Process Management System (BPMS)
- Presence and Leave Management (PLM)
- Ticketing (Service Desk)

All three follow the same patterns:

1. Pages expose task-oriented UI
2. A clear data model with well-known keys and relations
3. Role-based visibility and action authorization
4. Events and workflows that integrate across webbaselets

B.1 Common Design

Navigation. Each webbaselet registers under its own Area (e.g., slugs `bpms`, `plm`, `tickets`); pages include Dashboard, Browse/Detail, and Administration.

Security. Every action checks roles and ownership. Typical roles include: reader, contributor, approver, manager, and admin. Actions are logged.

Search and SEO. Content uses `keywords` and `description` attributes for in-context search. Lists expose filters and faceted navigation.

Auditability. All state transitions are captured with who/when/what; attachments preserved with versions.

Interoperability. Events are published as notifications that other webbaselets can subscribe to (e.g., Ticket created -> BPMS starts a triage workflow; Leave approved -> Ticket auto-closes).

B.1.1 Minimal Area Stub

A minimal root for each webbaselet as an STWArea:

```
{
  "_id": "area-bpms-root",
  "type": "Area",
  "namespace": "bpms",
  "name": { "en": "BPMS" },
  "slug": { "en": "bpms" },
  "children": []
}
```

Listing B.1: Minimal Webbaselet Root (Area)

B.2 Structured Classification for Webbaselets

To make webbaselets discoverable and governable across the ecosystem, the root `STWArea` of each webbaselet SHOULD declare a compact, structured *classification*. We recommend a dedicated `class` object on the Area to avoid collisions with core keys; engines MAY also surface these as indexed metadata for search.

B.2.1 Classification keys

Recommended fields and value ranges:

Field	Type	Purpose / Examples
purpose	enum	Primary intent: bpms, plm, ticketing, cms, directory, reporting, integration, admin.
domain	enum	Business area: hr, it, finance, sales, marketing, ops, legal, support, general.
capability	string[]	Features: workflow, approvals, forms, queueing, sla, search, analytics, files, notifications, calendar.
layer	enum	Architectural layer: ui, service, integration, data.
audience	string[]	Users: employees, managers, admins, agents, external.
criticality	enum	low, medium, high, critical.
maturity	enum	experimental, beta, stable, deprecated.
visibility	enum	public, authenticated, role-restricted.
compliance	string[]	gdpr, hipaa, sox, pci, iso27001, etc.
dataSensitivity	enum	public, internal, confidential, pii, phi.
i18n	string[]	Supported locales (BCP 47): e.g., ["en","it"].
engine	object	Runtime bounds: "min": "1.3.0", "max": "<2.0.0".
dependsOn	string[]	Other webbaselets by namespace and range, e.g., ["directory>=1.2", "notifications"].
imports	string[]	Subscribed events/exports (namespaced), e.g., ["bpms.task.completed"].
exports	string[]	Emitted events/components, e.g., ["tickets.created", "ui.widget:calendar"].
tags	string[]	Free-form keywords for discovery.

B.2.2 Example

BPMS classification on its root Area:

```
{
  "_id": "area-bpms-root",
  "type": "Area",
  "namespace": "bpms",
  "name": { "en": "BPMS" },
  "slug": { "en": "bpms" },
  "class": {
    "purpose": "bpms",
    "domain": "hr",
    "capability": ["workflow","approvals","forms","sla"],
    "layer": "ui",
    "audience": ["employees","managers","admins"],
    "criticality": "high",
    "maturity": "stable",
    "visibility": "authenticated",
    "compliance": ["gdpr"],
    "dataSensitivity": "pii",
    "i18n": ["en","it"],
```

```
    "engine": { "min": "1.3.0" },
    "dependsOn": ["directory>=1.1", "notifications"],
    "imports": ["tickets.created"],
    "exports": ["bpms.task.created","bpms.task.completed"],
    "tags": ["process","tasks","orchestration"]
  },
  "children": []
}
```

Listing B.2: STWArea with classification

Notes

- Engines should index `class.*` for search and dependency checks.
- Recommending enums keeps catalogs consistent; use `tags` for additional nuance.
- Version ranges follow semantic versioning; omit `max` for open-ended support.

B.3 Ecosystem Webbaselet (Community Catalog)

The Spin the Web Portal (see part III) includes an *Ecosystem* webbaselet that catalogs community-created webbaselets. Its purpose is to classify, make discoverable, and enable quality signals (rating/reviews) for webbaselets across the ecosystem. It builds directly on the structured classification in appendix B.2 and extends it with an opinionated, organization-based taxonomy.

B.3.1 Organization-based taxonomy

Classification aligns to common enterprise departments to aid discovery by function:

Domain	Examples
hr	onboarding, leave, compensation, training
it	service desk, change, assets, access management
finance	expenses, invoicing, budgeting, approvals
sales	leads, opportunities, quotes, forecasts
marketing	campaigns, content hub, events, assets
ops	facilities, fleet, logistics, maintenance
legal	contracts, policies, compliance
procurement	vendors, sourcing, purchase requests
support	knowledge base, ticket deflection, CSAT
rnd	requirements, roadmaps, experiments
product	releases, feature flags, feedback
security	incidents, awareness, audits
admin	directory, identity, governance
general	dashboards, search, navigation, utilities

These values map to `class.domain` and inform facets in catalog pages.

B.3.2 Data model

Key entities managed by the Ecosystem webbaselet:

```
{
  "catalogItem": {
    "_id": "eco-wbl-hr-directory",
    "namespace": "hr-directory",
    "name": {"en":"HR People Directory"},
    "summary": {"en": "Org-wide searchable people directory"},
    "class": {
      "purpose": "directory",
      "domain": "hr",
      "capability": ["search","profiles"],
      "tags": ["people","org","contact"]
    },
    "version": "1.2.0",
    "author": {"name":"Community Team","url":"https://spintheweb.org"},
    "links": {"repo":"https://...","docs":"https://...","demo":"https://..."},
    "status": "published"
  },
  "review": {
    "_id": "rev-123",
    "item": "eco-wbl-hr-directory",
    "user": "u:alice",
    "stars": 5,
    "comment": "Great starter for HR portals",
    "createdAt": "2025-09-12T10:00:00Z"
  },
  "rating": { "item": "eco-wbl-hr-directory", "average": 4.6, "count": 28 }
}
```

Listing B.3: Ecosystem entities

B.3.3 Workflows

- Submission: contributor provides metadata, classification, and links; item enters *submitted* state.
- Review/Moderation: approvers validate classification, security notes, and licensing; states: *approved*, *rejected*, *changes-requested*.
- Publish/Deprecate: approved items are listed; subsequent versions can supersede; deprecated items remain searchable with badges.
- Rating/Reporting: authenticated users rate (1–5) and review; abuse reports trigger moderation.

B.3.4 Representative pages

- Catalog: faceted browse by `domain`, `purpose`, `capability`, and tags; sort by rating and freshness.
- Item Detail: metadata, installation notes, screenshots, version history, ratings, related items.
- Submit/Manage: guided submission, preview, and moderation queue (admin).

B.3.5 Ratings and quality signals

- 1–5 star ratings with weighted average; first-party installs may carry a *verified* badge.
- Anti-spam: one rating per account per version; optional cool-down windows; abuse reporting.
- Signals: popularity (installs), freshness (recent updates), completeness (docs, tests), compliance tags.

B.3.6 Integration

- Events: `ecosystem.item.published`, `ecosystem.item.updated`, `ecosystem.review.created`.
- Dependencies: Directory (profiles), Notifications, Search/Index.
- Portal linkage: featured items and curated collections surfaced on spintheweb.org (see part III).

B.4 BPMS Webbaselet

Purpose: model, execute, and monitor business processes with tasks, SLAs, and approvals.

B.4.1 Core Concepts

Process Named workflow definition with versioning
Instance Runtime execution of a Process
Task Unit of work assigned to a role/user
Event Signal that triggers transitions or external actions

B.4.2 Data Model

```
{
  "process": { "_id": "proc-id", "name": {"en": "Onboarding"}, "version": 3, "stages":
    ↪ ["init","docs","it","done"] },
  "instance": { "_id": "pi-123", "process": "proc-id", "stage": "docs", "state": "active",
    ↪ "assignees": ["hr:manager"], "due": "2025-09-30" },
  "task": { "_id": "task-789", "instance": "pi-123", "type": "approve", "owner":
    ↪ "hr:manager", "status": "open", "slaHours": 48 },
  "event": { "kind": "task.completed", "ref": "task-789", "by": "hr:manager", "at":
    ↪ "2025-09-12T08:30:00Z" }
}
```

Listing B.4: BPMS entities

B.4.3 Representative Pages

- Dashboard: My Tasks, Overdue, SLA breaches
- Process Designer: stage/transition editor (admin)
- Instance Detail: timeline, attachments, comments

B.4.4 WBLL Snippet

```
\s('caption="My Tasks" key="taskId"')
lf l f l f // id, type, status
```

Listing B.5: My Tasks list (sketch)

B.5 PLM Webbaselet

Purpose: track presence, leave requests, balances, holidays, and approvals.

B.5.1 Data Model

```
{
  "calendar": { "year": 2025, "holidays": ["2025-01-01","2025-12-25"] },
  "balance": { "user": "u:alice", "year": 2025, "vacation": 15, "sick": 5 },
  "leave": { "_id": "lv-42", "user": "u:alice", "from": "2025-09-15", "to": "2025-09-20",
    ↪ "type": "vacation", "status": "pending", "approver": "mgr:bob" }
}
```

Listing B.6: PLM entities

B.5.2 Representative Pages

- My Calendar: presence, requests, approvals
- Team Calendar: manager view, conflicts
- Balances: accrual, consumption, carryover

B.5.3 Events

Examples: `leave.requested`, `leave.approved`, `leave.rejected`. Ticketing can subscribe to create handover tickets; BPMS can trigger onboarding/offboarding.

B.6 Ticketing Webbaselet

Purpose: log, track, and resolve issues/requests with queues, priorities, and SLAs.

B.6.1 Data Model

```
{
```

```
  "ticket": { "_id": "t-1001", "queue": "it", "kind": "incident", "priority": "high",
    ↪ "status": "open", "subject": "Laptop fan noisy", "requester": "u:carol",
    ↪ "assignee": "it:agent01" },
  "comment": { "_id": "c-1", "ticket": "t-1001", "by": "u:carol", "when":
    ↪ "2025-09-12T09:00:00Z", "text": "Happens after 10min" },
  "sla": { "queue": "it", "kind": "incident", "responseHours": 4, "resolutionHours": 48 }
}
```

Listing B.7: Ticket entities

B.6.2 Representative Pages

- Submit Ticket: form with WBLL inputs, attachments
- My Tickets: list, filters, bulk actions
- Agent Console: triage, assign, merge, close, macros

B.6.3 Cross-Webbaselet Integrations

- BPMS: auto-create approval tasks for change requests
- PLM: auto-snooze tickets when requester is on leave
- Directory: enrich requester profile and permissions

B.7 Roles and Permissions

Role	Capabilities
reader	browse lists and details
contributor	create/update own domain entities
approver	review/approve pending items
manager	oversee queues/teams, reports
admin	configure processes, queues, calendars

B.8 Notes on Deployment

Each webbaselet can be versioned and deployed independently. Use namespace scoping, feature flags, and migration scripts for data evolution. Provide sample datasets for demos and QA.

Bibliography

[1] Business process model and notation (bpmn), version 2.0.2. Technical Report formal/13-12-09, Object Management Group, January 2014. Standard specification.

[2] Thomas Allweyer. *BPMN 2.0: Introduction to the Standard for Business Process Modeling*. BoD–Books on Demand, 2 edition, 2016.

[3] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Universal resource identifiers in www. *Request for Comments*, 1630, 1994.

[4] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2 edition, 2018.

[5] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[6] Jakob Freund and Bernd R"ucker. *Real-Life BPMN: Using BPMN 2.0 to Analyze, Improve, and Automate Processes in Your Company*. Camunda Services GmbH, 2 edition, 2014.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[8] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, 2015.

[9] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814, 2008.

[10] Bruce Silver. *BPMN Method and Style*. Cody-Cassidy Press, 2 edition, 2011.

[11] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, B. Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[12] W3C. Xml schema part 1: Structures second edition. *W3C Recommendation*, 2004.

[13] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. Json schema: A media type for describing json documents, 2020.