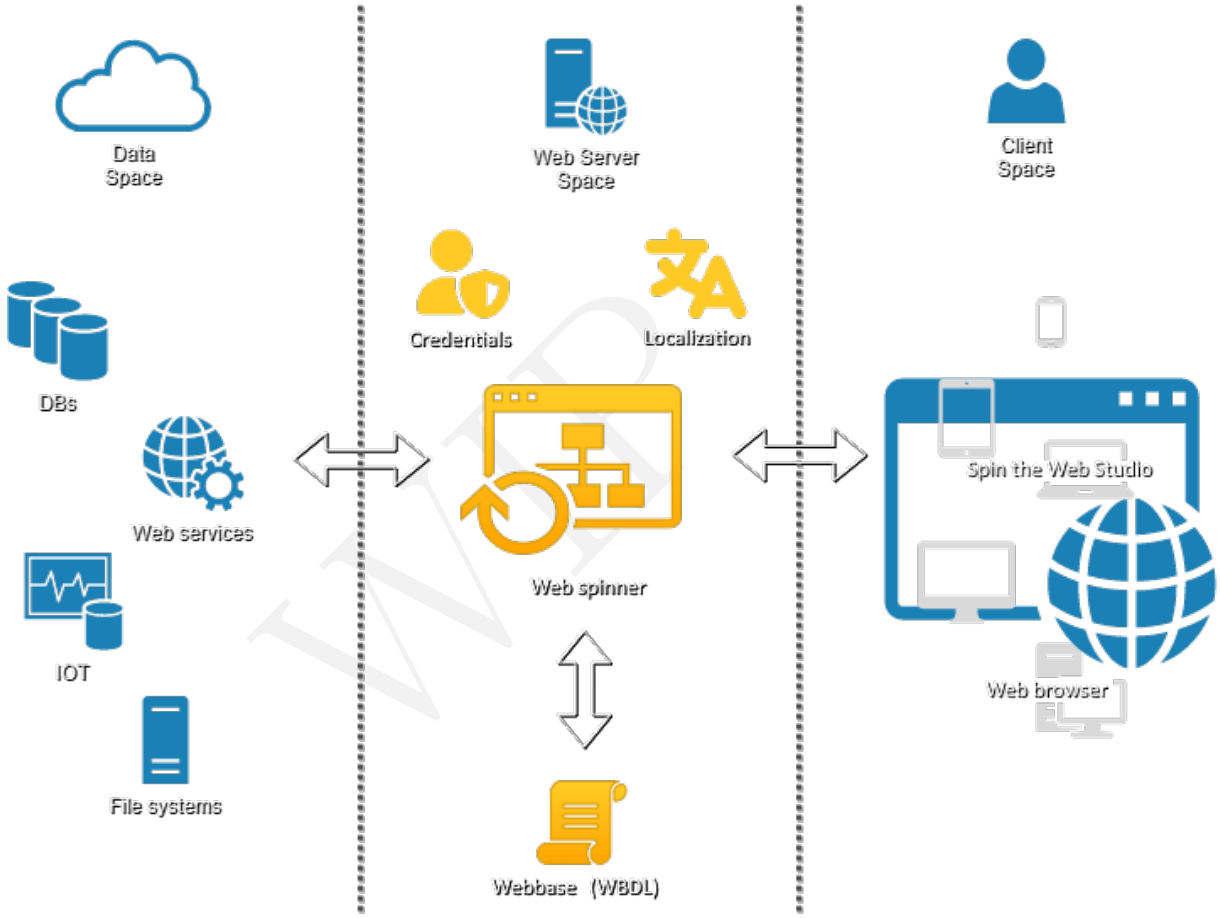
# Spin the Web

Weaving Web Portals




Spin the Web Project

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

*Compiled on 2026-02-22 17:14:51+01:00*

# License

*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 an open source framework for building **Enterprise Web Portals** ("portal"), with the intent of virtualizing the enterprise. In this book, a portal is a single access point that unifies the capabilities of websites and web applications: it aggregates and personalizes information, exposes interactive and transactional workflows, and serves as the enterprise brand's primary digital harbor. 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 objective is not to replace but to harmonize: envision a future where software vendors focus on product logic and expose APIs, while specialized UI integrators build the user interfaces on top of them. The framework is stewarded by the Spin the Web Project.

There are three core components:

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

The framework advances the **Virtualized Enterprise 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 combines foundational concepts with practical guidance for developers, integrators, and technology leaders seeking to modernize digital channels.

**Keywords:** enterprise web portals; ebranding; webbase; webbaselets; framework; Spin the Web Project; STW.

# 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 only at the data level, let's integrate them also 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* runtime 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 enterprise"—a single, coherent digital interface that adapts to each user's role[s] 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 eBranded.

## 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 webbased 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** establishes the theoretical foundations, explaining the problem space and introducing the core concepts of Spin the Web Framework.

**Part II** dives deep into the Framework: the `WBDL` language specification, the *Web Spinner* runtime architecture, and the *Spin the Web Studio* development environment.

**Part III** focuses on 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 Project

The mission of the Spin the Web Project is to divulge, manage, and evolve the Spin the Web framework while keeping it free to use. The Project 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 Project.

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—a journey that dates back to 1998.

Giancarlo Trevisan

2026-02-22

# Contents

# Part I

# The Foundations

# Introduction to Part I: The Foundations

> *"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 enterprise"—a unified digital interface that adapts to each stakeholder's role[s] 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* runtime for dynamic content delivery, and the *Spin the Web Studio* for development and maintenance.

**Chapter 1: Genesis and History** (§ 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** (§ 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 Enterprises** (§ 3) – Introduces the concept of portals as "virtualized enterprises"—unified digital interfaces that provide role-based access to all organizational functions, serving diverse stakeholders from employees to customers to partners.

**Chapter 4: Architecture Overview** (§ 4) – Provides an overview of the core architecture that makes the virtualized enterprise vision possible: the `WBDL` specification, the *Web Spinner* runtime, and the *Spin the Web Studio* development environment.

These foundational concepts lay the groundwork for the technical exploration presented in subsequent parts of the book.

# Chapter 1

# Genesis and History

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 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 enterprise represented the complexity typical of many organizations: 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 enterprise'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, Lotus 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 Lotus 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 browser-based 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

[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.

- **Intuitive Data Interaction**: Clients could inspect data intuitively and perform insertions, updates, and deletions (CRUD operations)
- **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 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 enterprise to build a site—an eSite, later termed a **portal**—whose target audience extended far beyond the general public.

This portal would serve:

- **Enterprise 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, webbased 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 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 portals 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 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 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, an enterprise, 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 **portal** whose target audience encompasses:

- **Public**: General public and media
- **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
- **Developers**: Technical contributors and integrators
- **Other 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 contents
**Data Sources** : Connections to databases
**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 paradigms

### 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 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 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 framework presented in this book represents several years 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 § 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 enterprise"—a single, coherent digital channel for all business interactions.[1]

It is important to note that Spin the Web is a professional framework. It is not a low-code/no-code platform for the general public, but rather a toolset for engineers building bespoke, enterprise-grade web portals. On this subject we could open a parenthesis: low-code/no-code platforms often promise ease of use and rapid development but frequently fall short when it comes to scalability, customization, and integration with existing systems. Spin the Web aims to fill this gap by providing a robust framework that empowers developers to create highly tailored solutions without being constrained by the limitations of low-code/no-code environments.

[1] Enterprises like SAP should prioritize robust data structures and business logic, exposing them through well-documented APIs rather than enforcing rigid user interfaces. While their ambition to "include the world" is commendable, the resulting environments often suffer from visual incoherence and operational friction with the enterprise's unique needs. *Spin the Web* advocates a modular approach: let foundational systems focus on data integrity and process orchestration, while role-based portals handle presentation, interaction, and organizational identity. See Sections 2.4 and 2.7 for a deeper exploration of this architectural separation.

## 2.2 Understanding Web Portals

The development of an enterprise 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.

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 enterprises**, allowing individuals, based on their specific role[s], to interact with every facet of the organization—from administration and logistics to sales, marketing, human resources, and production. It is a comprehensive framework 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 enterprise 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 portals or external systems, which can interact with it programmatically

`WBDL` is the language specifically designed to model the complexity of 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 enterprise ***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 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 command 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 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 enterprise 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 enterprise, 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 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 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 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 portal development prioritizes stakeholder experience and organizational coherence, viewing the portal not as a collection of features but as an 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 portal truly become a "virtualized enterprise"?

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 Enterprises

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

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

## 3.1 Understanding Virtualized Enterprises

A virtualized enterprise is more than just a website, a traditional web portal or even a web application. 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.

Web portals functioning as virtualized enterprises 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[s] 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 enterprise portal must serve a much more diverse audience:

### 3.2.1 Internal Stakeholders

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

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

**Employees** : Access to task management, communication tools, enterprise 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 enterprise 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 enterprise 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 enterprise 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 command performance

## 3.5 Benefits of the Virtualized Enterprise Model

Organizations that successfully implement virtualized enterprise 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 enterprise 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 Enterprise'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 enterprise's **quality management principles and manuals**. It is the living embodiment of the corporate soul.

By defining not only *what* an enterprise does but also *how* it does it, the portal transforms abstract procedural documents into interactive, enforceable workflows. It ensures that the enterprise'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 enterprise.

# Chapter 4

# 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 portals. This architecture aligns to three book parts for coherence: the Foundations, the Framework, and the Portal. 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 portal structures, data relationships, and business logic

**Web Spinner Runtime** : A runtime 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 Foundations: 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 Framework: Web Spinner

The Web Spinner runtime serves as the execution layer 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 Module** : Evaluates role-based visibility rules in real-time

**Command Processor** : Handles `WBPL` placeholder resolution and datasource command execution

**Content Cache** : Manages caching strategies for improved performance

**Session Manager** : Maintains user state and authentication information

### 4.3.3 Role-Based Visibility (Overview)

Spin the Web enforces authorization using a *role-based visibility* (RBV) model. Rather than returning explicit authorization errors, elements not visible to the requester are treated as non-existent and yield a `200 OK` with an empty response. This privacy-by-design approach applies uniformly to contents, pages, areas, and the site.

Key characteristics:

- **Declarative rules in `WBDL`**: visibility is defined per element and can be inherited.
- **Inheritance**: when a role setting is `undefined`, visibility is inherited recursively from the parent up to the root.
- **Three-state roles**: `true` (granted), `false` (denied), `undefined` (inherit).
- **Safe defaults**: unauthenticated users receive the `guests` role by default; elements are invisible unless made visible via rules or inheritance.

For operational details see § 9.3 in the Web Spinner chapter, and for the formal schema see the `WBDL` appendix (§ A).

**Side note: `rwx` and RBV** The familiar UNIX-style `rwx` access paradigm can be expressed with RBV: a content that supports reading, writing, or executing (in any combination) can be exposed to specific roles. In practice, "read" aligns with viewing content, while "write" and "execute" align with content operations (e.g., mutations or actions). RBV governs *visibility* of these capabilities per role; the concrete enforcement of operation-level permissions is handled at runtime by the Web Spinner security layer (see § 9.3).

### 4.3.4 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 Portal: 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

Foundations, Framework, and Portal 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 runtime 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 runtime
2. The runtime 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 Foundations, Framework, and Portal 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. We will explore the complete specifications of the languages and the architectural details of the Web Spinner runtime.

# Part II

# The Framework

# Introduction to Part II: The Framework

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

This part opens the hood of the Spin the Web framework, examining its specifications. We will dissect the core components introduced in the architecture overview: the languages that serve as the framework's instruction set, the modular design that enables scalability, and the mechanics of the engine that brings it all to life.

**Chapter 5: The WBDL Language** (§ 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** (§ 6) – Explores the Webbase Placeholders Language, which enables dynamic content injection and template-based portal generation.

**Chapter 7: The WBLL Language** (§ 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** (§ 8) – Examines the modular component system that allows for reusable portal elements and cross-platform integration.

**Chapter 9: The Web Spinner** (§ 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** (§ 10) – Introduces the integrated development environment for building and managing webbases.

**Chapter 11: Technology Stack and Implementation** (§ 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 specifications, runtime, and tools.

# Chapter 5

# Webbase Description Language (WBDL)

> *"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 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 `command` attributes in `STWContent` elements, where it replaces placeholders with actual values from various sources before command 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. JSON Schema definitions for `WBDL` elements are in § A (see Appendix: WBDL JSON Schema Reference).

### 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 portal. It inherits from `STWElement` and acts as the root element of the portal structure.

#### 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 command 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
- Command 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`.

#### 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.

### 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.

#### 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.

**command** : The command to be executed against the specified data source. Before execution, the command text is processed by the **Webbase Placeholders Language (`WBPL`)** processor. This processor replaces placeholders within the command with values sourced from several locations: the `params` attribute, the URL querystring, session variables, global variables, and HTTP headers. After this substitution, the resulting command 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 command, formatted as a standard querystring (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 functionalities:

- Shortcuts are contents that point to other contents. It's a form of content reuse.
- Client-side code are execution blocks (JavaScript)

- Server-side code are execution blocks (API)

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.

#### 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 command 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 Command 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 `command` 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 querystring 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
- Command 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 *command* 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 command 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 (§ 8) and the runtime (§ 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 B.

# 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 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 command 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

# The Web Spinner

> *"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 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 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 Security

Security in Spin the Web encompasses both **authentication** and **authorization**.

- **Authentication** verifies that an entity is who it claims to be.
- **Authorization** determines whether an authenticated entity can access a given service.

### 9.3.1 Authentication

Spin the Web supports native authentication via:

- **User/password pairs**
- **Service-based credentials** (e.g., API tokens or external identity providers)

Authenticated entities are assigned one or more roles.

**Initial Role Assignment** When a portal is accessed, the requester is automatically treated as a **guest user**.

- The guest user is assigned the `guests` role by default.
- This role governs visibility for unauthenticated users.
- Content not explicitly visible to `guests` will return a `200 OK` with an empty response.

This ensures safe default behavior and prevents metadata leakage.

### 9.3.2 Authorization: The Visibility Paradigm

Authorization in Spin the Web is based on **visibility**. Content is either:

- **Visible** → served to the requester
- **Invisible** → treated as non-existent, returning a `200 OK` with an empty response

This paradigm simplifies access control and enforces privacy by design.

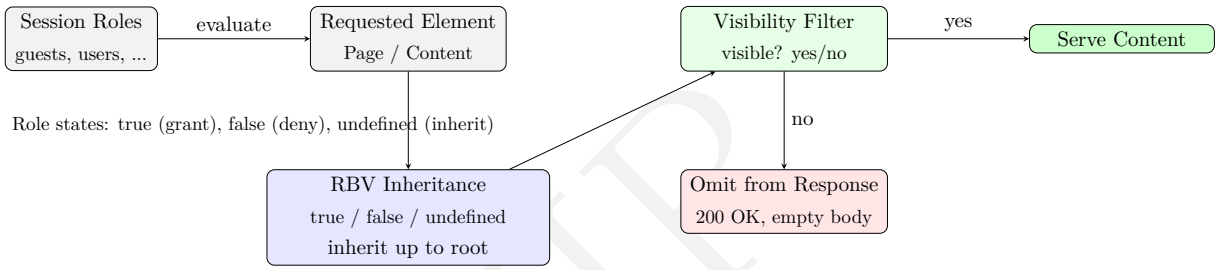



**Figure 9.1:** Role-Based Visibility flow: roles are evaluated against element RBV with inheritance; invisible elements are omitted without error.

**Role States** Each role can be in one of three states:

**`true`** explicitly granted
**`false`** explicitly denied
**`undefined`** inherited from parent elements

Inheritance follows a recursive model from the content up to the site root.

**Role Assignment** Roles can be assigned to:

- Individual contents
- Pages
- Areas
- Entire site

If no role is explicitly assigned, it inherits from its parent.

**Default Role: Guests** The `guests` role is automatically assigned to the portal home page and its contents. All other elements are invisible to guests unless explicitly made visible.

### 9.3.3 Predefined Roles

Spin the Web defines the following non-deletable roles:

- `roots` *superuser; see override below*
- `guests`
- `users`
- `administrators`
- `developers`
- `webmasters`
- `translators`

You may define additional roles as needed. A good starting point is your organizational chart.

**Superuser override (`roots`)** The `roots` role represents a trusted superuser. When a session includes `roots`, the "no assignment found up to the site level" condition does *not* cause a deny; instead, evaluation proceeds to the element's `visible` flag. This does not change the semantics of `visible=false` (which still hides the element).

### 9.3.4 RBV Decision Procedure

With reference to Figure 9.1, the Web Spinner applies Role-Based Visibility (RBV) using the following precise rules. Let *roles* be the set of roles in the current session; visibility flags are tri-state {`true`, `false`, unset}, where unset is treated as `true`.



#### Page decision

1. If the requested page is the Home page, **ALLOW** by default.
2. Else, if no authorization for any of the session roles is found on the page or any ancestor (page → area → site), **DENY** — *unless* the session has the `roots` role, in which case continue to step 3.
3. Else, if `page.visible` is `false`, **DENY** and *navigate to the Home page.*
4. Otherwise, **ALLOW**.

#### Content decision (for each content on an allowed page)

1. If the content or its ancestors (page → area → site) define authorizations for the session roles:
   (a) If none of those authorizations evaluate to `true`, **DENY**.
   (b) Else, if `content.visible` is `false`, **DENY**.
   (c) Otherwise, **ALLOW**.
2. If no role assignment for any session role is found all the way up to the site level, treat as an *explicit deny* — *unless* the session has the `roots` role, in which case evaluate `content.visible`: if `false` then **DENY**, otherwise **ALLOW**.

#### Defaults and inheritance

- Pages are *default-deny* (unless Home), requiring an explicit or inherited allow for at least one session role, and `visible` must not be `false`. If no role assignment is found all the way to the site level, treat the outcome as an *explicit deny*; sessions with the `roots` role bypass this specific condition and proceed to the `visible` check.
- Contents without any role assignment up to the site level are *explicitly denied*. Sessions with the `roots` role bypass only this condition and proceed to the `visible` check.
- RBV inherits up the hierarchy; evaluation short-circuits at the nearest level that defines role entries. If none are found by the time the site is reached, the result is an explicit deny (subject to the `roots` override above).

## 9.4 User Session Management

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

### 9.4.1 Session Establishment

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

### 9.4.2 Authentication Integration

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

### 9.4.3 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.

### 9.4.4 Session State

The session maintains:

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

## 9.5 Request Routing and Processing

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

### 9.5.1 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"

### 9.5.2 Element Resolution

Using the pre-built routing table, URLs are resolved to specific `WBDL` elements. Runtime outcomes:

- **Pages**: If the URL resolves to a page that is not visible to the current session (per Figure 9.1 and § 9.3.4), the user is taken to the **Home page**. If the path itself is invalid, the nearest site or area main page is served.
- **Contents**: If a content is not visible, it is omitted entirely and the request returns `200 OK` with an empty body. If the path is invalid, no result is returned.

### 9.5.3 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

## 9.6 Visibility and Authorization Engine

Before any content is delivered, the Web Spinner performs comprehensive authorization checks. See § 9.3 for the visibility model and predefined roles; this section focuses on runtime enforcement:

### 9.6.1 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

### 9.6.2 Dynamic Filtering

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

### 9.6.3 Secure Response Generation

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

## 9.7 Content Request and Response Cycle

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

### 9.7.1 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

### 9.7.2 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

### 9.7.3 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 command
3. Processes the command through the `WBPL` (Webbase Placeholders Language) engine
4. Executes the processed command against the datasource
5. Applies the content layout transformation
6. Returns the rendered content or raw data (depending on the request type)

## 9.8 Datasource Connection and Command Management

The Web Spinner maintains a sophisticated datasource management system:

### 9.8.1 Connection Pooling

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

### 9.8.2 Command 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** : Command plans may be cached for frequently-executed queries

### 9.8.3 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 command languages (MongoDB, Elasticsearch, etc.)
**File Systems** : Direct file access and processing

### 9.8.4 Error Handling

Datasource errors are gracefully handled:

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

## 9.9 Performance Optimization and Timeout Management

The Web Spinner implements several performance and reliability features:

### 9.9.1 Content Timeouts

Each content element has configurable timeout settings:

**Command Timeout** : Maximum time allowed for datasource queries
**Layout Timeout** : Maximum time for layout compilation and rendering
**Total Request Timeout** : Overall timeout for content delivery

### 9.9.2 Caching Mechanisms

Multiple levels of caching improve performance:

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

### 9.9.3 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

### 9.9.4 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.

## 9.10 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* runtime offers a robust execution layer, 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 webbased 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* runtime 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 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 Command 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 development environment necessary to move from Foundations to Framework to working Portal. 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 framework—from the declarative languages to the runtime 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 runtime 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.

Other implementations are welcomed, provided they adhere to the specifications and interoperability guidelines set forth by the Spin the Web Project.

## 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)
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 interpreter:

- 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

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

## 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* interpreter (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 runtime 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 portals with the platform we have just described.

# Part III

# The Portal

# Introduction to Part III: The 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 now divides implementation into two complementary dimensions:

- **Portal Contents (Structure, Semantics, Information)** — Modeling Areas, Pages, and Content blocks; embedding documentation; designing navigation, search, and task-oriented information architecture (§ 12).
- **Portal Visuals (Presentation, Layout, Interaction)** — Translating structure into accessible, brand-aligned, performant user interfaces while preserving semantic integrity (§ 13).

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 § C.3.

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—treating structure and presentation as orthogonal layers that evolve independently yet harmonize in the delivered experience.

# Chapter 12

# Implementing Portal Contents: Structure, Semantics, and Information Flow

*"Structure is the substrate of meaning; when well designed it becomes self-documenting and accelerates delivery."*

This chapter covers how to model and implement the *informational side* of an enterprise portal using `WBDL`: Areas, Pages, and Content blocks as semantic anchors; embedded documentation as a living knowledge system; and patterns for navigation, search, and task execution. The emphasis is on how structure transforms raw enterprise data into durable, navigable knowledge assets.

## 12.1 From Organization to Information Architecture

The starting point for portal content modeling is the enterprise itself. Business functions, departments, service domains, and stakeholder-facing capabilities provide the *semantic inventory* that becomes your top-level `STWArea` set. Mapping is a translation exercise:

1. Collect the official organizational chart(s) and cross-functional process maps.
2. Identify external-facing vs. internal-only domains; decide which are public, private, or hybrid.
3. Normalize naming (short, role-recognizable labels) and prune redundancies.
4. Annotate each candidate Area with primary audiences, core data domains, and governing policies.
5. Prioritize rollout sequence (critical operations first, supporting domains next, exploratory later).

**Example Top-Level Areas** A manufacturing enterprise might define: Sales, Administration, Backoffice, Technical Office, Products & Services. Each becomes an `STWArea` root for subordinate Pages and Content.

Area definitions embed *living documentation* through localized `name`, `description`, and `keywords`. This collapses the distance between operational knowledge and runtime structure: the portal itself becomes the manual.

## 12.2 Hierarchical Semantics and Namespaces

Hierarchy provides cognitive compression. The *Hierarchical Namespace Pattern* organizes content so users predict where information lives. Recommended principles:

- Depth over width only when each level adds semantic clarity.
- Keep sibling counts manageable ($7 \pm 2$ heuristic for primary navigation tiers).
- Use stable identifiers (slugs) for Areas/Pages to preserve deep links and audit trails.
- Reflect role segmentation (what differs by role) at the `STWContent` or conditional rendering layer rather than duplicating structural nodes.

Roles shape *visibility* and *interaction surfaces* but should not fork the structural namespace unnecessarily. This separation sustains maintainability and consistent analytics.

## 12.3 Documentation Through Structure

Every `WBDL` element (Site, Area, Page, Content) hosts localized `description` and `keywords`. These fields form a vertically integrated documentation system that can encode: quality policies, compliance references, process identifiers, KPIs, audit obligations, data classification tags, and internal taxonomy terms. Table 12.1 summarizes scope alignment.

**Site** Mission, global quality policy, enterprise standards.
**Area** Departmental procedures, regulatory scope, operating KPIs.
**Page** Process walkthroughs, workflow prerequisites, exception handling.
**Content** Field-level rules, validation, data retention, privacy notes.

Embedding this metadata enables: contextual help, machine-assisted authoring, internal semantic search, automated compliance extraction, and AI augmentation (linking processes to data touchpoints).

**Table 12.1:** Documentation scope by structural level

| Level | Primary Focus | Typical Metadata | Example Questions Enabled |
|---|---|---|---|
| Site | Enterprise Mission | Mission, quality policy, audit standards, global KPIs | What is the overarching mission? Which global standards govern all Areas? |
| Area | Department / Domain | Procedures, regulatory scope, data domains, risk class | Which regulations apply to this domain? Who owns the process? |
| Page | Process / Journey | Workflow steps, prerequisites, exception handling, SLA | What happens before/after this step? What are escalation paths? |
| Content | Field / Action Unit | Validation rules, retention, privacy flags, policy codes | Can this value be exported? How long is data retained? |

The metadata lattice supports contextual help, semantic search ranking, compliance extraction, and AI-assisted authoring.

## 12.4 Core Page Archetypes

Across domains three archetypes dominate main content regions:

- **Dashboards**: Multi-source summaries for situational awareness and prioritization. Favor glanceable KPIs, deltas, and exception surfacing.
- **Tabular / List Views**: Filterable, paginated, sometimes hierarchical or pivot-capable collections enabling search+select workflows.
- **Detail Views**: Focused inspection/editing surfaces for a single entity with contextual navigation (previous/next, related entities, timeline overlays).

Composite pages combine archetypes (e.g., miniature dashboard widgets above a list). Resist uncontrolled aggregation—each archetype should answer a distinct question.

## 12.5 User Journeys and Page Design

Pages (`STWPage`) embody discrete user tasks or journeys. Design methodology:

1. **Task Enumeration**: Elicit core user goals per Area (create quote, approve order, analyze defect rate).
2. **Journey Mapping**: Sequence states and data intersections; identify required contextual jumps (entity cross-links) to minimize session fragmentation.
3. **Structural Allocation**: Assign each journey to a Page; merge only if cognitive load stays low and authorization scopes match.
4. **Section Layout**: Header (global actions), sidebars (navigation, filters, summaries), main body (primary archetype), footer (feedback, contacts).
5. **Instrumentation**: Embed identifiers and metadata for analytics, ticketing, and performance tracing.

**Integrated Ticketing** A mature portal treats structure itself as a support surface. Ticket submissions should auto-capture structural scope (site/area/page/content IDs), user locale, and active filters. This converts qualitative feedback into structured, triage-ready data.

## 12.6 Building with Content Blocks

`STWContent` instances are atomic functional or informational units: forms, tables, charts, grids, banners, maps, metrics, documents. Composition principles:

- **Single Responsibility**: Each block answers one primary question or supports one action cluster.
- **Declarative Data Binding**: Use `WBPL` for parameterized queries; keep transformation logic close to data retrieval for transparency.
- **Progressive Disclosure**: Lazy-load heavy or low-priority content blocks to reduce initial cognitive and performance load.
- **Embedded Governance**: Include validation rules, policy codes, and retention hints inside `description` metadata.

- **Reusability**: Parameterize commonly repeated patterns (e.g., entity summary panels) rather than cloning definitions.

**Example Composition** A public "Products" page may combine: hero banner (static), category menu (dynamic list), product grid (dynamic, filtered), featured carousel (dynamic curated), and call-to-action block—each a discrete `STWContent`.

## 12.7 Search, Discovery, and Internal SEO

Localized `keywords` and `description` fields drive internal search relevance. Two complementary modes:

**Global Search** Traverses full structural graph; returns heterogenous results ranked by metadata quality, usage signals, and recency.

**In-Context Search** Operates within the active Page scope: filters list/table rows, quick-finds form fields, queries localized help.

Design guidance: persistent global search affordance (often header); proximal in-context search control near target content. Provide keyboard shortcuts, accessible labeling, and query history. Treat internal search analytics as a requirements radar (unmet search intent indicates structural or content gaps).



## 12.8 Patterns and Evolution

Long-term iteration reveals recurring cycles: pattern recognition → insight → abstraction → simplification. Content modeling matures by:

- Consolidating duplicated structures into parameterized definitions.
- Elevating cross-cutting concerns (authorization tags, retention policies) into shared vocabularies.
- Refactoring deep hierarchies when navigation analytics show friction.
- Capturing emergent naming conventions to stabilize taxonomy.
- Instrumenting evolution (version history of structural nodes) to support audits and AI summarization.

The endpoint is elegant simplicity: a minimal, expressive structure that scales without combinatorial explosion.

## 12.9 Conclusion

The content layer transforms disparate enterprise data into structured, navigable, contextual information assets. By treating every node (Area/Page/Content) as both functional unit and documentation carrier, the portal becomes a continuously evolving, queryable knowledge base that supports operations, compliance, and learning.

# Chapter 13

# Implementing Portal Visuals: Presentation, Layout, and Interaction

*"Visual design should disappear into function, letting users feel the system rather than notice it."*

This chapter focuses on the *visual and interaction layer* of a Spin the Web portal: translating semantic structure into perceivable, accessible, performant, and brand-aligned experiences—without compromising the stability of information architecture defined in § 12. The goal is to make presentation an *orthogonal dimension*: evolvable, themable, and testable in isolation.

## 13.1 Separation of Structure and Presentation

Structure lives in `WBDL`: Areas, Pages, and Content blocks define semantics, navigation graph, and documentation. Presentation lives in a theming and layout layer (CSS variables, design tokens, component library primitives) that consumes structural metadata. This separation yields:

- **Predictable Evolution**: New visual themes deploy without structural migrations.
- **Stable Deep Links**: URLs and identifiers remain constant across redesigns.
- **Semantic Integrity**: Search, analytics, and AI augmentations operate on durable structure, not brittle selectors.
- **Test Focus**: Functional tests target structure and logic; visual regression suites target presentation surfaces.

Anti-patterns: encoding layout meaning into identifiers; duplicating Pages solely for cosmetic variants; baking brand colors into content definitions.

**Design Tokens** Use an invariant token vocabulary (e.g., `color.surface.raised`, `space.xs`, `font.heading.scale`) mapped to concrete theme values. Tokens enable multi-brand deployments and dark/light adaptation with minimal churn.

## 13.2 Layout Regions and Responsiveness

Canonical regions: Header (global nav, identity, search), Primary Navigation (lateral or horizontal), Context Pane (filters, related entities), Main Content, Ancillary Panels (notifications, collaboration), Footer (legal, contact). Responsive strategy principles:

1. **Mobile First Density**: Collapse non-critical side regions into overlay drawers; preserve primary task path above decorative brand elements.
2. **Breakpoint Semantics**: Define breakpoints where interaction model changes (hover removed $<$ tablet, multi-column grids collapse).
3. **Progressive Disclosure**: Defer rendering of off-screen heavy panels until first invocation.
4. **Adaptive, Not Just Responsive**: Personalize ordering of blocks based on role usage analytics while retaining structural IDs.

Use CSS grid/flex abstractions aligned to tokenized spacing; avoid ad hoc pixel constants.

## 13.3 Navigation Systems and Flow

Navigation manifests user mental models of the structure. Layers:

**Global Navigation** Exposes top-level Areas; must be stable, sparse, and keyboard navigable.

**Local Navigation** Contextual to an Area or Page section (tab sets, side menus) for lateral moves.

**Breadcrumbs** Encode path memory; avoid redundancy when global navigation already fully mirrors hierarchy depth 1.

**Inline Relational Links** Provide graph exploration (related entities, recent activity) without forcing full context switch.

Design heuristics: prefer *predictable repetition* (consistent placement) over novelty; minimize hover-only affordances; provide explicit focus order. Instrument navigation events to surface friction (abandoned path segments, loop behaviors).

**Image and Iconographic Navigation** Navigation can also be accomplished through **clickable images**, floor-plan style "planimetries", diagrams, or semantically meaningful icons. These affordances are powerful for spatial, facilities, product catalog, or process topology exploration. To integrate them without sacrificing accessibility or scalability:

- **Semantic Redundancy**: Every image-driven affordance must have a parallel text link or ARIA-labelled control to support screen readers and search indexing.
- **Vector Over Raster**: Prefer SVG with distinct, focusable regions (using `<a>` or `role="link"`) instead of image maps on raster graphics; enables theming and high-DPI crispness.
- **Descriptive Tooltips**: Provide concise tooltips / `title` attributes; avoid encoding primary meaning solely in color or pictogram shape.
- **Responsive Hit Targets**: Ensure minimum target size (44 px CSS) across breakpoints; adapt layout so targets do not cluster too densely on touch devices.
- **Keyboard Pathways**: Maintain logical tab order across hotspot regions; allow arrow-key or WASD navigation for spatial grids where appropriate.

- **Progressive Enhancement**: Initial load should render a textual list fallback while SVG/hotspot script enhancements hydrate.
- **Performance Budget**: Large planimetric diagrams should be chunked or lazily loaded; defer non-visible layers (e.g., annotations) until interaction.
- **Analytics Instrumentation**: Tag each hotspot with structural IDs (Area/Page/Content) and business semantics to measure navigation efficacy and dead zones.
- **Scalability**: For dynamic catalogs, auto-generate hotspot layers from the `WBDL` structure rather than manually editing coordinates.
- **Theming**: Bind fills, strokes, and hover states to design tokens (not hard-coded colors) for dark mode and brand adaptations.

Use image-based navigation to *accelerate discovery*, not to replace canonical hierarchical paths. Spatial or iconographic maps become a complementary entry surface feeding into the same stable URL and breadcrumb system.

## 13.4 Dashboards, Tables, and Details

Visual treatments align to archetype intent:

- **Dashboards**: Prioritize contrast, succinct labeling, and anomaly highlighting. Limit cardinality of primary KPI tiles ($7 \pm 2$). Provide consistent tile grammar: label, value, delta, trend.
- **Tables / Lists**: Support density toggles (comfortable/compact), sticky headers, column personalization. Emphasize scannability: left-align textual identifiers, right-align numeric aggregates, reserve iconography for actionable states.
- **Detail Views**: Anchor with an identity bar (title, status, primary actions). Segment content with visual rhythm (spacing tiers) not heavy borders. Provide contextual next/previous navigation when part of a result set.

Cross-cutting: embed loading placeholders shaped like final content (skeletons) to reduce perceived latency; ensure empty states teach next action.

## 13.5 Role-Based Visual Adaptation

Authorization governs data visibility; visual adaptation communicates scope without fragmenting structure. Techniques:

- Conditional display of Content blocks (retain layout placeholders to avoid jumpy reflow when possible).
- Progressive disclosure of advanced filters and bulk actions only for privileged roles.
- Declarative role tags bound to CSS utility classes for subtle emphasis (e.g., manager-only highlights).
- Unified audit overlay mode (toggle) to display why elements are hidden (for debugging and governance).

Avoid: duplicating Pages per role; mixing security logic into visual components instead of central policy engines.

## 13.6 Feedback, Ticketing, and Observability

Integrate a frictionless feedback loop aligned with structure (see § 12 on instrumentation). Key UI constructs:

- Inline issue trigger (icon/button) scoped to Page or Content block; auto-captures structural IDs, user locale, and viewport size.
- Session timeline panel exposing recent navigation, feature flags, and performance spans for support replication.
- Visual state indicators: latency spinners, deferred load badges, stale data warnings with last refresh timestamp.
- Observability overlay (admin only) highlighting render cost hotspots / API latency zones.

Close the loop: a ticket success toast with deep link to status; encourage user re-engagement instead of dead-end submission pages.

## 13.7 Search Experience Design

Two surfaces (global, in-context) must *feel* unified. Global search: omnipresent field or icon opening a command palette style modal; supports keyboard invocation, fuzzy matching, and result type grouping. In-context search: embedded within tables, forms, or navigation panels, scoping results to current structural node.



#### Design Principles

- Immediate focus when opening global search; preserve prior query history with non-intrusive ghost text.
- Provide semantic filters (Area/Page) as chips; avoid forcing users to learn query DSL initially.
- Highlight matched substrings for learnable relevance model feedback.
- Offer keyboard preview navigation with real-time detail pane (no full navigation until commit).

Accessibility: ARIA roles (combobox, listbox), announce result counts, maintain logical tab order returning user to invoking control when dismissed.

## 13.8 Accessibility and Inclusive Design

Inclusive design is multiplicative: it improves efficiency for all users while enabling access for those with specific needs. Core practices:

- Semantic HTML foundation; avoid `div`-only components.
- Color contrast meeting WCAG 2.1 AA (4.5:1 text, 3:1 large); test dark/light parity.
- Full keyboard operability: visible focus, logical order, skip links for main content.
- Motion reduction honoring prefers-reduced-motion; substitute subtle fades for large parallax.
- Localization readiness: mirrored layouts for RTL, flexible space for longer strings, numeric/date formatting.

- Assistive hints in metadata (descriptions) powering contextual help and screen reader summaries.

Adopt automated accessibility testing plus manual audits (screen reader smoke tests) in CI.

## 13.9 Branding and Themability

Brand expression emerges from typography scale, color system, shape language (radius, stroke), and motion curve palette. Implementation strategy:

1. Define neutral base theme (grayscale, spacing, layout primitives).
2. Layer brand-specific tokens (color, typography, motion) referencing base tokens—no hard-coded hex in components.
3. Provide runtime theme switch (light/dark/high-contrast) with persisted user preference.
4. Guard legibility: run contrast validation when updating token sets; reject failing combinations.
5. Expose a Theme Inspector webbaselet for administrators to preview token diffs live.

Multi-brand portals map different token sets to identical `WBDL` structures, enabling re-use of governance and search analytics.

## 13.10 Performance and Perceived Speed

User perception hinges on *time to meaningful interaction*. Strategies:

**Skeleton Screens** Mirror final layout; avoid generic spinners that reset user mental model.

**Incremental Data Hydration** Stream primary entities first; defer secondary panels (analytics, recommendations).

**Optimistic UI** Reflect user intent immediately (button state, provisional row) with reconcile fallback if server rejects.

**Edge Caching** Cache static assets and token manifests at CDN; version tokens for instant theme rollbacks.

**Resource Budgeting** Define performance budgets (kb per route, API latency SLOs) and surface violations in build pipeline.

Instrument Web Vitals (LCP, INP, CLS) and map to structural IDs for precise remediation. Performance data should be browsable through an internal observability webbaselet.

## 13.11 Conclusion

A disciplined visual layer preserves semantic clarity while amplifying usability, trust, and brand presence. By keeping presentation orthogonal to structure, portals remain evolvable: new themes, devices, and interaction models can emerge without refactoring the underlying `WBDL` definitions. Visual excellence compounds: accessible, performant, token-driven interfaces reduce cognitive load and accelerate enterprise adoption.

# Part IV

# The Roadmap

# Introduction to Part IV: The Roadmap

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

The chapter that follows, *The Roadmap* (§ 14), collects the most promising threads and defines criteria for promotion into the main body of the book.

# Chapter 14

# The Roadmap

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 enterprise maintains a structured 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 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 enterprise's portal could automatically command 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

[sectionUnification of WBDL and AI: Consequences and Opportunities]

The unification brought about by `WBDL` and modern AI has profound consequences for the future of digital ecosystems:

- **Semantic Interoperability:** `WBDL` provides standardized, machine-readable descriptions of web resources. AI systems can leverage this structure to understand, reason about, and manipulate web content, enabling automation and intelligent integration across platforms.
- **Automated Reasoning and Decision Making:** AI agents can use `WBDL`'s structured data to perform complex reasoning, automate workflows, and make context-aware decisions, resulting in smarter web applications that adapt to user needs and business environments.
- **Accelerated Innovation:** Developers and organizations can build new tools and services more rapidly, as AI can interpret and act on `WBDL`-described resources without manual intervention, reducing development time and errors.
- **Enhanced Discoverability and Personalization:** AI systems can use `WBDL` metadata to better understand user intent and preferences, enabling more accurate search, recommendation, and personalization features.
- **Cross-Domain Integration:** The modularity of `WBDL` and the generalization capabilities of AI make it easier to connect disparate systems, data sources, and services, fostering a unified and intelligent web ecosystem.

In summary, the synergy between `WBDL` and AI transforms the web from a collection of static resources into a dynamic, intelligent, and interoperable environment, driving new possibilities for automation, personalization, and innovation.

## 14.8 API-First Enterprise Software: WBDL as Universal UI

Looking forward, the Project actively advocates for enterprise software vendors to provide robust, well-documented APIs capable of managing all aspects of their software. This is essential for global adoption of the API-first paradigm: organizations expose business processes, data, and operations through APIs—REST, GraphQL, or other standards—while WBDL acts as the universal UI layer, consuming these APIs to build flexible, unified portals.

**The Project's Position:** For Spin the Web to reach its full potential, enterprise software houses must offer comprehensive APIs for their products. These APIs should cover all major business functions, data access, and operational controls, enabling WBDL to consume and orchestrate them in portal UIs. This advocacy is essential for:

- Modernizing legacy and monolithic systems without rewriting core logic
- Integrating diverse platforms into unified, role-based user experiences
- Accelerating portal development and deployment
- Future-proofing organizations against backend migrations or upgrades
- Enabling machine-to-machine business communication and automation

In this paradigm, WBDL orchestrates UI, navigation, and user journeys, while all backend interactions—authentication, business rules, data queries—are handled via API calls. This separation of concerns supports agile development, scalability, and global adoption. As more enterprise systems adopt API-first strategies, WBDL can unify them, delivering consistent, brand-aligned experiences for both human and machine users.

Crucially, WBDL can interact with any enterprise software—provided the data is made accessible—using a uniform paradigm: data is searched, the search result is a list, and the items in the list can be explored. This means that on a single webpage, different datasources (from multiple systems) can be accessed and presented as if they were managed by a single unified software. This capability opens the full universe of eBranding: organizations can design digital experiences that seamlessly blend data and functionality from diverse platforms, delivering a coherent brand identity and user journey across all business domains.

Intelligent agents enhance the portal experience rather than replace it. WBLL and WBDL remain the foundation, while agents act as advanced collaborators—able to read, write, and reason across the webbase to accelerate meaningful work and unlock new possibilities for automation and insight.

# Appendices

# Appendix A

# WBDL JSON Schema Reference

This appendix collects the formal JSON Schema definitions for the main WBDL element types used in Spin the Web. These schemas provide the canonical structure for sites, areas, pages, and content elements.

## A.1 STWElement Base

The abstract base for all WBDL objects. Provides identity, common metadata (localized name, slug, keywords, description), visibility, and an optional recursive `children` array used by composed types. Concrete types specialize via `type`.



**Properties**

**`_id` (string, uuid) (required)** Global unique identifier for the element instance.
**`type` (enum)** Discriminator for concrete kinds: `"Site" | "Area" | "Page" | "Content"`.
**`name` (STWLocalized)** Localized display name used in navigation and titles.
**`slug` (STWLocalized)** Localized, URL-safe identifier used to compose paths.
**`keywords` (STWLocalized)** Search keywords and tags aiding discovery.
**`description` (STWLocalized)** Human-readable summary for listings or previews.
**`visibility` (STWVisibility)** Role based boolean visibility.
**`children` (array of STWElement)** Optional recursive structure for nested compositions.

```
{
  "$id": "https://spintheweb.org/schemas/STWElement.json",
  "$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 A.1:** STWElement Base Schema Definition

## A.2 STWLocalized

Represents a localized string as an array of language–text pairs, e.g., `["lang":"en","text":"Hello", "lang":"en-US","text":"Color", "lang":"it","text":"Ciao"]`. Use IETF BCP 47 language codes (language only like `en` or language+region like `en-US`).

```
{
  "$id": "https://spintheweb.org/schemas/STWLocalized.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWLocalized",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "lang": { "type": "string", "description": "IETF BCP 47 language tag (e.g., 'en',
    ↪ 'en-US')" },
      "text": { "type": "string" }
    },
    "required": ["lang", "text"],
    "additionalProperties": false
  },
  "minItems": 1
}
```

**Listing A.2:** STWLocalized Schema Definition

```
[
  { "lang": "en",    "text": "Color" },
  { "lang": "en-US", "text": "Color" },
  { "lang": "en-GB", "text": "Colour" },
  { "lang": "it",    "text": "Colore" }
]
```

**Listing A.3:** STWLocalized Examples (language and region-qualified tags)

## A.3 STWSite

Represents the root of a portal. Inherits common fields and introduces: the `mainpage` (landing page UUID), optional `langs` list, optional `datasources`, and `children` Areas. Use this as the top-level container for navigation and localization scope.

### Properties

**`type` (const "Site")** Discriminator for this shape.

**`mainpage` (string, uuid) (required)** UUID of the portal's landing page (homepage).

**`langs` (array of string)** List of supported language codes (IETF BCP 47), e.g., `"en"`, `"en-US"`. The first in the list is the default; if omitted, the array defaults to `["en"]`.

**`datasources` (array of object)** Integration endpoints/configuration objects.

```
{
  "$id": "https://spintheweb.org/schemas/STWSite.json",
  "$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" },
      "default": ["en"]
    },
    "datasources": {
      "type": "array",
      "items": { "type": "object" }
    }
  },
  "required": ["mainpage"]
}
```

**Listing A.4:** STWSite Schema Definition

**Minimum webbase** Because `STWSite.mainpage` is required and points to the homepage, the smallest valid webbase consists of one `STWSite` and one `STWPage` whose UUID matches `mainpage`.

```
{
  "_id": "b1b8c9fa-3d9e-4c44-8fd6-5a8436cfe8a1",
  "type": "Site",
  "name": [
    { "lang": "en", "text": "Spin the Web" },
    { "lang": "it", "text": "Spin il Web" }
  ],
```

```
  "slug": [
    { "lang": "en", "text": "spin-the-web" },
    { "lang": "it", "text": "spin-il-web" }
  ],
  "mainpage": "f47ac10b-58cc-4372-a567-0e02b2c3d479",
  "langs": ["en", "en-US"],
  "children": []
}
```

**Listing A.5:** ]STWSite Example with explicit langs ["en", "en-US"]

```
[
  {
    "_id": "0c2a7f6b-1d3e-4a5b-8c9d-0e1f2a3b4c5d",
    "type": "Site",
    "name": [ { "lang": "en", "text": "Demo Site" } ],
    "slug": [ { "lang": "en", "text": "demo-site" } ],
    "mainpage": "9a9c3b6e-1d9a-4e62-9e10-0b25e84684ef",
    "children": [
      {
        "_id": "9a9c3b6e-1d9a-4e62-9e10-0b25e84684ef",
        "type": "Page",
        "name": [ { "lang": "en", "text": "Home" } ],
        "slug": [ { "lang": "en", "text": "home" } ],
        "children": [
          {
            "_id": "6f1a1a52-1fa6-4a03-9e38-8d80d1b6c29b",
            "type": "Content",
            "subtype": "Text",
            "name": [ { "lang": "en", "text": "Greetings" } ],
            "slug": [ { "lang": "en", "text": "greetings" } ],
            "section": "main",
            "layout": { "lang": "en", "text": "Hello World!" }
          }
        ]
      }
    ]
  }
]
```

**Listing A.6:** Minimal Webbase: STWSite + homepage STWPage

## A.4 STWArea

A sectional grouping below the Site. Typical uses include functional domains (e.g., Support, Sales) or topic silos. Areas can be modeled in two ways:

- **Full Area**: Inherits `STWElement`, requires `mainpage` (UUID). Best when the area participates in site-wide navigation and indexing.
- **Lightweight Area**: Minimal proxy with only `src` (URI) to a webbaselet. Best for externally hosted or federated content that should appear as a single hop in navigation.

**Properties** *Common*: See `STWElement` for shared fields (`_id`, `name`, etc.).
*Full Area-specific*: `type` (const "Area"), `mainpage` (string, uuid) (required).
*Lightweight Area-specific*: `src` (string, uri) (required) pointing to the webbaselet.

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

**Listing A.7:** STWArea Schema Definition (two shapes via oneOf)

## A.5 STWPage

A concrete navigable page. Inherits common fields and constrains `children` to `STWContent` items in display sequence. Use Pages to compose content blocks and layouts.

### Properties

**`type` (const "Page")** Discriminator for this shape.

**children (array of STWContent) (required)** Ordered content blocks composing the page.

```
{
  "$id": "https://spintheweb.org/schemas/STWPage.json",
  "$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 A.8:** STWPage Schema Definition

## A.6 STWContent

An atomic content block rendered within a Page region. Supports optional `subtype`, CSS hooks, and data bindings (`dsn`, `command`, `params`); requires a `layout` describing placement.



### Properties

**type (const "Content")** Discriminator for this shape.
**subtype (string)** Content specialization hint (e.g., `"Form"`, `"Table"`, `"Calendar"`). Defaults to `"Text"`.
**cssClass (string)** Optional CSS class hook for theming and targeting.
**section (string)** Logical region name (e.g., `"main"`, `"aside"`).
**sequence (number)** Display order within the `section` (can be fractional, e.g., 10.5).
**dsn (string)** Data source name or connection identifier.
**command (string)** Operation to execute against the data source.
**params (string)** Serialized parameters for the command.
**layout (STWLayout) (required)** Placement and sizing information in the page grid.

```
{
  "$id": "https://spintheweb.org/schemas/STWContent.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "STWContent",
  "allOf": [{ "$ref": "STWElement.json" }],
  "properties": {
    "type": { "const": "Content" },
    "subtype": { "type": "string", "default": "Text" },
    "cssClass": { "type": "string" },
    "section": { "type": "string" },
```

```
    "sequence": { "type": "number" },
    "dsn": { "type": "string" },
    "command": { "type": "string" },
    "params": { "type": "string" },
    "layout": { "$ref": "STWLayout.json" }
  },
  "required": ["layout"]
}
```

**Listing A.9:** STWContent Schema Definition

## A.7 STWContentWithOptions

A content variant that references a set of available options by UUID (e.g., menus, tabbed panels, harmoniums) and selects among them via presentation logic; still requires a `layout`.

### Properties

**`type` (const "Content")** Discriminator for this shape.
**`subtype` (string)** Variant hint for presentation logic.
**`options` (array of string uuid) (required)** Identifiers of selectable option items.
**`layout` (STWLayout)** Placement and sizing information; required for rendering.

```
{
  "$id": "https://spintheweb.org/schemas/STWContentWithOptions.json",
  "$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 A.10:** STWContentWithOptions Schema Definition

# Appendix B

# WBLL Token Reference

This appendix provides a complete reference for WBLL tokens.

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

Brackets denote optional arguments, for example:

- No arguments: `e`
- One argument: `e('format')`
- Two arguments: `e('format;name')`
- Three arguments: `e('format;name;value')`

If you need to skip an 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): `rent`, `username`, `city`.

| rent | username | city |
|---|---|---|
| 450.00 | alice | Milan |
| 525.00 | bob | Paris |
| 400.00 | carol | Berlin |

*Assumptions for examples:* unless stated otherwise, (1) examples evaluate the first row; (2) the field cursor starts at the first field (`rent`); (3) when currency output like "€ 1,234.50" appears, it is the formatted `rent` 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**

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

---

```
<label>rent</label>450.00<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**

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

---

```
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**

```
>>><f
```

---

```
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**

```
>f
```

---

```
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**

```
// 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')
```

---

```
<tr class="highlight">... <input type="text" name="rent" value="450.00"> ...</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**

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

---

```
<input type="text" name="rent" value="450.00" required style="color:red"><input
    ↪ type="text" name="username" value="alice" maxlength="20">
```

| **Token** | `\r` |
|---|---|
| **Description** | Inserts a line break: `<br>`. Non-editing; produces no value output and does **not** advance the field cursor. |

**Example**

```
f\rf
```

---

```
450.00<br>alice
```

| **Token** | `e('format[;name][;value]')` |
|---|---|
| **Description** | Renders an input element. Arguments are *positional but optional*. Defaults: *format* = `text`, *name* = active field name, *value* = active field value. **Format options:** any HTML5 input type (`text`, `email`, `url`, `tel`, `number`, `date`, `datetime-local`, `time`, `month`, `week`, `color`, `range`, etc.); or a numeric/date-time formatting string (e.g., `#,##0.00` for decimals, `yyyy-MM-dd` for dates). Numeric patterns set `type="number"` with appropriate `step` attribute. **Cursor rule (editing):** advances when *name* is not specified; if *name* is provided, does not advance (even if *value* is omitted). **Non-interactive mode:** renders as **f** (formatted field value). |

**Example**

```
e('email;userEmail') // HTML5 email input type
e('date;startDate') // HTML5 date input
e('#,##0.00;rent;450.00') // Numeric format pattern -> type="number" step="0.01"
e // Defaults to type="text", uses active field
```

---

```
<input type="email" name="userEmail" value="450.00">
<input type="date" name="startDate" value="alice">
<input type="number" name="rent" value="450.00" step="0.01" inputmode="decimal">
<input type="text" name="rent" value="450.00">
```

| **Token** | `f('format')` |
|---|---|
| **Description** | Renders the active field value at the cursor, optionally applying a *format*. The *format* argument is positional but optional (default: identity). **Cursor rule (non-editing):** advances because it consumes the active field value. **Formatting:** applies to numeric and date-time values (e.g., `#,##0.00` for decimals, `yyyy-MM-dd` for dates). |

**Example**

```
f('#,##0.00')\r // Numeric formatting of active field
f('yyyy-MM-dd')\r // Date formatting
f // No format (identity)
```

---

```
1,234.50<br>2025-11-30<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**

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

---

```
<input type="hidden" name="rent" value="450.00"><input type="hidden" name="kind"
    ↪ value="area"><input type="hidden" name="id" value="alice">
```

| **Token** | `j('code')` |
|---|---|
| **Description** | Inserts a JavaScript code block as `<script>code</script>`. If *code* is omitted, uses the active field value. Placeholders (e.g., `@@placeholder`) within *code* are automatically replaced with their values. Use `\@` to escape the `@` symbol if literal text is needed. **Cursor rule (non-editing):** advances only when *code* is empty (consuming active field value). |

**Example**

```
// assuming @@rows = 3, active field value = "alert('Hello');"
j('function foo() { alert("what?"); }')
j('console.log("Rows:", @@rows)')
j('// Email: user\@example.com')
j>
```

---

```
<script>function foo() { alert("what?"); }</script>
<script>console.log("Rows:", 3)</script>
<script>// Email: user@example.com</script>
<script>alert('Hello');</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**

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

---

| **Token** | `l('id[;text]')` |
|---|---|
| **Description** | Renders a label element *as a sibling* (not wrapping) to the next input element: `<label for="<id>"><text></label>`. The `for` attribute references the next input's `id`. Arguments: *id* (optional, defaults to auto-generated ID), *text* (optional, defaults to active field name or value). **Cursor rule (non-editing):** does not advance. **Note:** Use uppercase **L** to wrap the next input inside the label. |

**Example**

```
l('uname;Username:')e('username')
```

---

```
<label for="uname">Username:</label><input type="text" name="username" id="uname"
    ↪ value="450.00">
```

| **Token** | `L('text')` |
|---|---|
| **Description** | Renders a label element that *wraps* the next input element: `<label><text> <input...></label>`. The *text* argument is optional (default = active field value or empty). Unlike lowercase **l**, this creates a label without `for` attribute and contains the input. **Cursor rule (non-editing):** does not advance. |

**Example**

```
L('Username:')e('username')
```

---

```
<label>Username: <input type="text" name="username" value="450.00"></label>
```

| **Token** | `m('format[;name][;value]')` |
|---|---|
| **Description** | Renders a multiline input. Arguments are *positional but optional*. **Format modes:** empty or `textarea` = plain `<textarea>`; `wysiwyg` = Summernote WYSIWYG editor; language code (`javascript`, `html`, `css`, `bash`, etc.) = Ace code editor with syntax highlighting. Defaults: *format* = textarea, *name* = active field name, *value* = active field value. **Cursor rule (editing):** advances when *name* is not specified; if *name* is provided, does not advance. **Non-interactive mode:** renders as **f** (formatted field value). |
| **Example** | |

```
m(';comments;Hello') // Plain textarea
m('wysiwyg;description') // Summernote editor
m('javascript;code;console.log("test")') // Ace editor with JavaScript syntax
```

---

```
<textarea name="comments">Hello</textarea>
<textarea id="..." name="description">alice</textarea><script>...</script>
<div id="..." class="stwCodeeditor">...</div><textarea id="..." name="code"
    ↪ style="display:none">console.log("test")</textarea><script>...</script>
```

| **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** | |

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

---

```
<table data-key="fId">
  <caption>Users</caption>
  <thead>Found: 3</thead>
  <tbody>
    <tr>
      <th>rent</th><td>450.00</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**

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

---

```
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**

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

---

```
<input type="password" name="pwd" value="secret"><br><input type="password"
    ↪ name="rent" value="450.00"><br><input type="password" name="username"
    ↪ value="password">
```

| | |
|---|---|
| **Token** | `a('url')p('name[;value]')t('text')` |
| **Description** | Renders an anchor element: `<a href="url?params">text</a>`. The *url* argument is optional (default = active field value). Use **p()** tokens to append query parameters; use **t()** or another display token for link text. **Cursor rule (non-editing):** advances only if *url* is empty (consuming active field value). **Note:** Use uppercase **A** to open link in new tab. |

**Example**

```
a('/users')p('id;@@userId')t('View User')
a>t('Link from field')
```

---

```
<a href="/users?id=123">View User</a>
<a href="450.00">Link from field</a>
```

| | |
|---|---|
| **Token** | `A('url')p('name[;value]')t('text')` |
| **Description** | Renders an anchor element with `target="_blank"`: `<a href="url?params" target="_blank">text</a>`. Same as lowercase **a** but opens in a new tab/window. **Cursor rule (non-editing):** advances only if *url* is empty. |
| **Example** | |

```
A('https://example.com')t('External Link')
```

---

```
<a href="https://example.com" target="_blank">External Link</a>
```

| | |
|---|---|
| **Token** | `p('name[;value]')` |
| **Description** | Appends a query parameter to the preceding **a**, **A**, **b**, or **o** token. Arguments: *name* (required), *value* (optional, defaults to active field value or placeholder if `@@name`). Does not produce output directly; modifies the URL/href of the parent element. **Cursor rule (non-editing):** advances only when *value* is empty (consuming active field value). |
| **Example** | See examples for **a**, **b**, **o** tokens. |

| | |
|---|---|
| **Token** | `b('url;namespace;action')p('name[;value]')t('text')` |
| **Description** | Renders a button or submit input. Arguments: *url* (optional redirect), *namespace* (typically `stw`), *action* (e.g., `insert`, `update`, `delete`, `search`, `filter`, `submit`, `logon`, `logoff`, `pwdreset`). Use **p()** for parameters, **t()** for button text. **Cursor rule (non-editing):** does not advance. |
| **Example** | |

```
b(';stw;insert')t('Add Record')
b(';stw;submit')t('Save')
```

---

```
<button type="submit" name="stwAction" value="stwinsert">Add Record</button>
<button type="submit" name="stwAction" value="stwsubmit">Save</button>
```

| Token | `c('mode;name;value;options...')` |
|---|---|
| **Description** | Renders checkbox input(s). **Option modes:** `mode=0` = datasource query (`dsn;SELECT ...`); `mode=1` = simple list (each item is both value and label); `mode=2` = key-value pairs (value;label;value;label...). For datasource mode, if query returns 1 field, acts as mode=1; if 2 fields (id, value), acts as mode=2. Arguments: *mode*, *name* (optional, defaults to active field), *value* (optional, defaults to active field value), then options. **Cursor rule (editing):** advances when *name* is not specified. **Non-interactive mode:** renders as disabled checkboxes. |

**Example**

```
c('0;tags;;mysqldb;SELECT id,name FROM tags') // Datasource (2 fields -> key-value)
c('1;agree;;yes;no;maybe') // Simple list
c('2;choice;;1;Option A;2;Option B') // Key-value pairs
```

---

```
<fieldset class="stwGroup"><label><input type="checkbox" name="tags" value="1">Tag
    ↪ One</label>...</fieldset>
<fieldset class="stwGroup"><label><input type="checkbox" name="agree"
    ↪ value="yes">yes</label>...</fieldset>
<fieldset class="stwGroup"><label><input type="checkbox" name="choice"
    ↪ value="1">Option A</label>...</fieldset>
```

| Token | `r('mode;name;value;options...')` |
|---|---|
| **Description** | Renders radio button input(s). **Option modes:** `mode=0` = datasource query (`dsn;SELECT ...`); `mode=1` = simple list; `mode=2` = key-value pairs. For datasource mode, if query returns 1 field, acts as mode=1; if 2 fields, acts as mode=2. Arguments: *mode*, *name* (optional), *value* (optional), then options. **Cursor rule (editing):** advances when *name* is not specified. **Non-interactive mode:** renders as disabled radio buttons. |

**Example**

```
r('1;gender;;male;female;other')
```

---

```
<fieldset class="stwGroup"><label><input type="radio" name="gender"
    ↪ value="male">male</label>...</fieldset>
```

| **Token** | `d('name[;value];mode;options...')` or `D(...)` |
|---|---|
| **Description** | Renders a dropdown select. **Lowercase d** allows empty selection (includes `<option></option>`); **uppercase D** requires selection (no empty option). **Option modes:** `mode=0` = datasource query (`dsn;SELECT ...`); `mode=1` = simple list; `mode=2` = key-value pairs. For datasource mode, if query returns 1 field, acts as mode=1; if 2 fields, acts as mode=2. Arguments: *name* (optional), *value* (optional), *mode*, then options. **Cursor rule (editing):** advances when *name* is not specified. **Non-interactive mode:** renders as **n** (mapped value display). |

**Example**

```
d('country;;0;mysqldb;SELECT id,name FROM countries') // Datasource
d('city;;1;Milan;Paris;Berlin') // Simple list
D('status;;2;1;Active;2;Inactive') // Key-value pairs
```

---

```
<select name="country"><option></option><option value="1">Italy</option>...</select>
<select name="city"><option></option><option value="Milan">Milan</option>...</select>
<select name="status"><option value="1">Active</option><option
    ↪ value="2">Inactive</option></select>
```

| **Token** | `s('name[;value];mode;options...')` or `S(...)` |
|---|---|
| **Description** | Renders a multi-select dropdown with `multiple` attribute. **Lowercase s** allows empty; **uppercase S** requires selection. **Option modes:** `mode=0` = datasource query (`dsn;SELECT ...`); `mode=1` = simple list; `mode=2` = key-value pairs. For datasource mode, if query returns 1 field, acts as mode=1; if 2 fields, acts as mode=2. **Cursor rule (editing):** advances when *name* is not specified. **Non-interactive mode:** renders comma-separated mapped values (similar to **n** but handles multiple selections). |

**Example**

```
s('tags;;1;tag1;tag2;tag3')
```

---

```
<select name="tags" multiple><option></option><option
    ↪ value="tag1">tag1</option>...</select>
```

| **Token** | `u` |
|---|---|
| **Description** | Renders a file upload input: `<input type="file">`. Uses active field name. **Cursor rule (non-editing):** does not advance. |

**Example**

```
u\a('accept="image/*"')
```

---

```
<input type="file" accept="image/*">
```

| **Token** | `i('mode;options...')` |
|---|---|
| **Description** | Renders an image element. **Modes:** empty or `0` = direct URL from *mode* or active field value; `1` = indexed by value (field value selects from option list); `2` = key-value pairs (field value matches key, renders corresponding URL). **Cursor rule (non-editing):** advances. |

**Example**

```
// assuming field value = 2
i('/img/logo.png')
i('1;img1.png;img2.png;img3.png')
i('2;1;img-one.png;2;img-two.png')
```

---

```
<img src="/img/logo.png">
<img src="img2.png">
<img src="img-two.png">
```

| **Token** | `n('mode;options...')` |
|---|---|
| **Description** | Displays a mapped value (non-interactive version of dropdown). **Modes:** `1` = indexed by value; `2` = key-value pairs. **Cursor rule (non-editing):** advances. |

**Example**

```
// field value = 2
n('1;Low;Medium;High')
n('2;1;Active;2;Inactive;3;Pending')
```

---

```
Medium
Inactive
```

| **Token** | `o('url')p('name[;value]')` |
|---|---|
| **Description** | Renders an article container element with auto-generated ID and optional href built from *url* and parameters. **Cursor rule (non-editing):** does not advance. |
| **Example** | |

```
o('/item')p('id')
```

---

```
<article id="..." href="/item?id=450.00"></article>
```

| **Token** | `v('expression')` |
|---|---|
| **Description** | Evaluates a JavaScript expression at compile time and inlines the result. **Warning:** executes at layout compilation time, not render time. Use with caution. **Cursor rule (non-editing):** does not advance. |
| **Example** | |

```
v('new Date().getFullYear()')
```

---

```
2025
```

| **Token** | `\n` |
|---|---|
| **Description** | Inserts a line break: `<br>`. Same as **\r**. Non-editing; does not advance the field cursor. |
| **Example** | |

```
t('Line 1')\nt('Line 2')
```

---

```
Line 1<br>Line 2
```

| Token | `\t` |
|---|---|
| **Description** | Inserts an empty label element: `<label></label>`. Useful for layout spacing in flex forms. Non-editing; does not advance the field cursor. |

**Example**

```
\tL('Name:')e
```

---

```
<label></label><label>Name: <input type="text" name="rent" value="450.00"></label>
```

| Token | `x('width')` |
|---|---|
| **Description** | Renders a horizontal bar element for plots and visualizations: `<span style="display:inline-block; width:<width>px; height:1rem;"></span>`. The *width* argument (in pixels) is optional; default = active field value. **Cursor rule (non-editing):** advances when *width* is not specified (consuming active field value). |

**Example**

```
// assuming active field value = 250
x('100')
x
```

---

```
<span style="display:inline-block; width:100px; height:1rem;"></span>
<span style="display:inline-block; width:250px; height:1rem;"></span>
```

| Token | `y('height')` |
|---|---|
| **Description** | Renders a vertical bar element for plots and visualizations: `<span style="display:inline-block; width:1rem; height:<height>px;"></span>`. The *height* argument (in pixels) is optional; default = active field value. **Cursor rule (non-editing):** advances when *height* is not specified (consuming active field value). |

**Example**

```
// assuming active field value = 200
y('150')
y
```

---

```
<span style="display:inline-block; width:1rem; height:150px;"></span>
<span style="display:inline-block; width:1rem; height:200px;"></span>
```

| **Token** | z |
|---|---|
| **Description** | Reserved token; currently no-op (produces no output). May be used for future extensions. Non-editing; does not advance the field cursor. |
| **Example** | No output. |

# Appendix C

# 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



## C.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).

### C.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 C.1:** Minimal Webbaselet Root (Area)

## C.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.

### C.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. |

### C.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 C.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.

## C.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 § C.2 and extends it with an opinionated, organization-based taxonomy.

### C.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.

### C.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 C.3:** Ecosystem entities

### C.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.

### C.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).

### C.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.

### C.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).

## C.4 BPMS Webbaselet

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

### C.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

### C.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 C.4:** BPMS entities

### C.4.3 Representative Pages

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

### C.4.4 WBLL Snippet

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

**Listing C.5:** My Tasks list (sketch)

## C.5 PLM Webbaselet

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

### C.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 C.6:** PLM entities

### C.5.2 Representative Pages

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

### C.5.3 Events

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

## C.6 Ticketing Webbaselet

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

### C.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 C.7:** Ticket entities

### C.6.2 Representative Pages

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

### C.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

## C.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 |

## C.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.

# Appendix D

# Webbaselet Use Cases

This appendix illustrates typical use cases for webbaselets: small, self-contained web applications that used to live in spreadsheets, ad-hoc scripts, or custom line-of-business tools. The goal is not to replace existing systems, but to harmonize them: software vendors expose stable APIs, and UI integrators compose webbaselets that sit on top of those APIs.

Despite the name, webbaselets are not necessarily simple or small. At one end of the spectrum they can model a single form or report; at the other end they can embody a complete ERP, CRM, or vertical line-of-business solution. The key distinction is not size, but the fact that they are modular, API-driven, and composable.



## D.1 Team Task Board

### Before: Spreadsheet To-Do List

Many teams maintain a shared spreadsheet to track tasks: columns for assignee, status, due date, and comments. Over time, these files proliferate ("Q1_tasks_final_v7.xlsx"), access control becomes informal, and simple workflows (e.g. notifications, filtering by user) require manual effort.

### After: Task Board Webbaselet

A task board webbaselet connects to a tasks API (or a simple JSON data source) and offers:

- Column-based views (e.g. *Backlog*, *In Progress*, *Done*).
- Per-user filters and saved views.
- Inline editing, without opening or emailing files.
- Optional integration with issue trackers or chat tools.

The underlying logic (task model, permissions, notifications) remains in the core product; the webbaselet focuses on UI and composition.

## D.2 Inventory and Stock Tracking

### Before: Warehouse Spreadsheet

Small warehouses or teams often track inventory in a spreadsheet: SKU, description, location, stock level, reorder threshold. There is no single source of truth, concurrent edits conflict, and basic queries ("show low stock by location") require manual filtering.

### After: Inventory Webbaselet

An inventory webbaselet consumes an inventory API (or database view) and provides:

- Search and filtering by SKU, category, or location.
- Highlighting of low-stock and out-of-stock items.
- Simple forms for stock adjustments and movements.
- Read-only views for stakeholders who should not edit data.

The stock logic and validation stay in the backend; the webbaselet offers a focused UI that can be embedded in portals or dashboards.

## D.3 Event Registration and Attendance

### Before: Sign-up Forms + CSV Exports

Event organizers often rely on generic form builders that export CSV files. They manually merge lists, send reminders, and track attendance in a spreadsheet with ad-hoc columns.

### After: Registration Webbaselet

A registration webbaselet connects to an events/participants API and enables:

- Branded sign-up forms backed by a structured data model.
- Live views of registration counts per session or ticket type.
- Check-in views optimized for tablets or mobile devices.
- Export to downstream systems (CRM, email marketing) via APIs.

Again, business rules (capacity limits, ticket types, pricing) live in the backend. The webbaselet tailors the UI for a specific event or organizer.

## D.4 Lightweight CRM and Contact Lists

### Before: Contact Spreadsheet

Sales or support teams maintain contact lists in spreadsheets: name, company, email, notes, last-contact date. Different teams fork their own copies, and no one has a consistent, up-to-date view of customer interactions.

### After: Contact Webbaselet

A contact webbaselet integrates with a CRM or customer API and offers:

- Unified views of contacts with filters for segment, owner, or lifecycle stage.

- Inline note taking and tagging.
- Contextual links to external systems (support tickets, billing, product usage dashboards).

The CRM remains the system of record; the webbaselet is a customized lens focused on one team's workflow.

## D.5 Timesheets and Time Tracking

### Before: Weekly Timesheet Templates

Consulting and development teams often email around weekly timesheet templates as spreadsheets. Employees fill them in, send them back, and someone consolidates the data manually for payroll or billing.

### After: Timesheet Webbaselet

A timesheet webbaselet talks to a time-tracking or billing API and provides:

- Simple weekly views per person, project, or client.
- Validation rules delegated to the backend (e.g. max hours, required fields).
- Role-specific views (employee, manager, finance).
- Export or synchronization to accounting systems.

The core product owns timesheet logic and compliance; the webbaselet delivers a focused UI that can evolve independently.

## D.6 Guest Registration via QR Code

### Before: Paper Forms and Ad-hoc Apps

Guest registration is often handled with paper sign-in sheets, generic form builders, or improvised tablet apps. Data ends up scattered across mailboxes and CSV exports; hosts lack a live view of who has arrived, and compliance requirements (e.g. GDPR consent, visitor badges, emergency lists) are handled informally.

### After: Guest Registration Webbaselet

A guest registration webbaselet focuses entirely on the arrival experience and visitor lifecycle:

- Pre-registration links sent by email, each with a unique QR code.
- A check-in view that scans the QR code and confirms arrival.
- Dynamic questions per visit type (e.g. NDA, health & safety).
- Instant notifications to hosts and reception staff.
- Secure integration with identity systems and visitor logs via APIs.

The underlying policies (who may enter, retention times, badge rules) remain in the core system; the webbaselet specializes the UI around QR-code–based registration and check-in.

## D.7 Employee Timesheets

### Before: Spreadsheet-based Timesheets

In many organizations, employees track working hours in shared spreadsheets or emailed templates. Each week, files are copied, renamed, and manually consolidated for payroll and billing. There is no single source of truth, approval workflows are ad hoc, and basic checks (e.g. missing entries, overtime rules) are performed by hand.

### After: Timesheet Webbaselet

A timesheet webbaselet is dedicated to capturing, reviewing, and exporting working hours:

- Weekly and monthly views per employee, project, or client.
- Validation of entries (required fields, maximum hours, holidays) delegated to backend policies.
- Manager approval flows (pending, approved, rejected) surfaced as simple UI states.
- Role-specific perspectives for employees, managers, and finance.
- API-based integration with HR, payroll, and invoicing systems.

The core HR or ERP system remains the system of record; the webbaselet provides a focused, evolvable UI for time capture and approval, instead of relying on fragile spreadsheet processes.

## D.8 Human Resources and Welfare Management

### Before: Fragmented HR and Welfare Processes

HR and welfare programs are often split across multiple tools: spreadsheets for benefits eligibility, email threads for requests, PDF forms for reimbursements, and separate portals for leave, training, and well-being initiatives. Employees struggle to find the right entry point, HR staff re-key data across systems, and visibility on participation and costs is limited.

### After: HR and Welfare Webbaselet

An HR and welfare management webbaselet unifies access to policies, requests, and status tracking:

- A single entry point where employees can view benefits, submit requests (e.g. childcare, wellness, commuting), and track approvals.
- Dynamic workflows based on role, location, and employment type, driven by backend rules.
- Integration with HRIS, payroll, and benefits providers via APIs, avoiding manual data transfer.
- Aggregated dashboards for HR to monitor utilization, budget impact, and compliance indicators.

The core HR and payroll systems keep ownership of contracts, eligibility, and calculations; the webbaselet orchestrates a coherent, employee-centric UI for welfare management at scale.

These examples illustrate a recurring pattern: existing spreadsheets and ad-hoc tools reveal stable data structures and workflows. Webbaselets capture those patterns as reusable, composable UI pieces that sit on top of well-defined APIs, harmonizing rather than replacing the underlying systems.

# Bibliography


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

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

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

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

[5] 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.

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

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

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

[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.

# Glossary

**API**

Application Programming Interface. A set of definitions and protocols for building and integrating application software. 133

**eBranding**

The practice of virtualizing all virtualizable aspects of an entity (organization, enterprise, trade, or group) into a single, coherent digital channel. eBranding unifies documentation and action: it presents public and private information, interactive workflows, and system integrations through a role-aware portal that acts as the brand's primary digital harbor. It emphasizes consistency, accessibility, and governance across audiences (public, customers, suppliers, partners, regulators, employees), and favors integration over replacement by orchestrating APIs and existing systems behind a unified experience. v, 9, 133

**portal**

An **Enterprise Web Portal**: a single access point that unifies the capabilities of a website (public information and brand presence), a traditional web portal and a web application (aggregation, personalization, discovery, interactive and transactional workflows). In this book, unless otherwise specified, the term *portal* refers to the **Enterprise Web Portal** and is capable of handling all virtualizable activities associated with an enterprise across roles and channels. v, 133

**REST**

Representational State Transfer. A software architectural style that defines a set of constraints to be used for creating Web services. 133

**URI**

Uniform Resource Identifier. A unique sequence of characters that identifies a logical or physical resource. 133

**webbase**

A complete WBDL-described portal: the full, declarative structure of a site including areas, pages, contents, datasources, and configuration. 133

**webbaselet**

A modular fragment of a webbase. A WBDL document rooted at an STWArea that can be imported or integrated into a full webbase. 133