Selver Softic

Introduction to the Basics of Web Technologies

Presa Universitară Clujeană

Selver Softic

Introduction to the Basics of Web Technologies

PRESA UNIVERSITARĂ CLUJEANĂ

2026

Scientific reviewers:

Assoc. Prof. Dr. Vlad Bocăneț
Technical University of Cluj-Napoca

Dipl.-Ing. Safet Softic
CAMPUS 02 University of Applied Sciences, Graz, Austria

Introduction to the Basics of Web Technologies
FH-Prof. DI Dr. techn. Selver Softic, BSc
CAMPUS 02 University of Applied Sciences,
Graz, Austria

ISBN 978-606-37-2971-3

Universitatea Babeș-Bolyai
Presa Universitară Clujeană
Director: Codruța Săcelean
Str. B.P. Hasdeu nr. 51
400371 Cluj-Napoca, România
Tel.: +40 744 687 884
E-mail: editura@ubbcluj.ro
www.editura.ubbcluj.ro | libraria.ubbcluj.ro

Content

Preface

Dear Readers,

This book, Introduction to the Basics of Web Technologies, provides a solid foundation in modern web technologies. From TCP/IP protocols and HTTP to HTML, CSS, and JavaScript, it explains the building blocks of the internet in a practical way – ideal for computer science students, web developers, and IT professionals.

In times of rapid digitalization, understanding these fundamentals is essential for cloud computing, web development, and digital transformation. This material draws from my teaching at CAMPUS 02 and reflects current standards like HTML5, IPv6, and GDPR.

Graz, March 2026

FH-Prof. DI Dr. techn. Selver Softić, BSc

Introduction

This book provides a comprehensive introduction to the technical pillars of web development and the Internet. It starts with the basics of the World Wide Web and explains the development and functioning of network protocols such as TCP/IP and addressing via IP addresses and the Domain Name System (DNS). A detailed section breaks down the components of Uniform Resource Identifiers (URIs) and discusses their application, including the risks of URL shorteners.

Furthermore, the sources are dedicated to frontend design through HTML and Cascading Style Sheets (CSS), where selectors, length units, box model, and various positioning approaches are explained. Finally, client-side logic is complemented by an introduction to JavaScript that focuses on the correct declaration of variables (const, let) and the various basic data types. An accompanying list of abbreviations also summarizes important terms from database and software architecture.

This manuscript is written for students, educators, engineers, and web designers, to provide them with some initial foundations for real-world applications.

Detailed Chapter Overview

The chapters of this book follow a logical progression from foundational Internet concepts to practical frontend development. Each chapter builds systematically on the previous one, creating a cohesive learning path for computer science students, web developers, IT professionals, and educators. This overview provides a comprehensive preview of the content, key learning objectives, practical applications, and interconnections between topics. By understanding this structure, readers can appreciate how individual technologies work together to form the modern web ecosystem.

Chapter 1: Basics of the Internet and the World Wide Web

This foundational chapter establishes the core concepts of the digital ecosystem that underpins all web technologies. It begins by distinguishing the Internet – the global "network of networks" providing physical and logical connectivity – from the World Wide Web, which represents one specific service layer built upon this infrastructure. Readers learn to differentiate between these often-confused terms and understand their respective roles in daily digital interactions, from e-learning platforms to cloud services.

The chapter details the TCP/IP protocol family, with particular emphasis on the differences between TCP (Transmission Control Protocol) for reliable, connection-oriented communication and UDP (User Datagram Protocol) for fast, connectionless transmission. Through comparative analysis and practical examples, students explore the four-layer TCP/IP model – from physical transmission to application protocols – and recognize why this stack has become the global standard due to its scalability, interoperability, and extensibility. This knowledge forms the technical basis for understanding all subsequent protocol discussions.

A comprehensive section on IP addressing covers both IPv4 and IPv6, explaining address exhaustion challenges, hierarchical allocation systems (IANA, RIRs, LIRs), private address ranges, and Network Address Translation. The Domain Name System (DNS) is presented as the "telephone book of the Internet," with detailed explanation of resolution processes, hierarchical structure, and record types. These concepts are crucial for web hosting, domain management, and network troubleshooting.

Modern web browsers are examined as the primary gateway to web content, including their core features, current market shares, and the critical role of W3C standardization in preventing "browser wars." The chapter concludes with resource identification (URI, URL, URN structures, query parameters, fragments) and security foundations (HTTPS, TLS 1.3), providing the complete picture of how users access and interact with web resources securely.

Learning objectives: Distinguish Internet infrastructure from WWW services; explain TCP/IP layers and protocol choices; navigate IP addressing and DNS resolution; understand browser standards and resource identification. **Applications**: Network configuration, web hosting, basic cybersecurity.

Chapter 2: Communication and Performance

Building directly on Chapter 1's protocols, this chapter examines the application layer's workhorse: HTTP and its evolution. The Hypertext Transfer Protocol is presented as the stateless client-server model enabling all web interactions, with detailed coverage of methods (GET, POST, PUT, DELETE), status codes, headers, and the request-response lifecycle. Readers learn why statelessness simplifies scalability but requires additional mechanisms for user sessions.

The core challenge of HTTP statelessness is addressed through HTTP cookies, examined from definition and operation through anatomy (name-value pairs, attributes like Domain, Path, HttpOnly, Secure, SameSite), classification (session vs. persistent, first-party vs. third-party), GDPR compliance requirements, and modern alternatives (LocalStorage, IndexedDB). This section emphasizes security implications and data protection best practices, reflecting current European regulations.

Performance optimization receives thorough treatment through web caching mechanisms (browser, proxy, CDN levels), invalidation strategies (TTL, ETags, conditional headers), and next-generation protocols (HTTP/2 multiplexing, HTTP/3 over QUIC). Students learn measurable techniques to reduce latency by 50% or more, essential for real-world applications like video streaming and e-commerce.

Learning objectives: Master HTTP methods and lifecycle; implement secure state management; apply caching and modern protocols for performance. **Applications**: Web server configuration, session handling, site optimization.

Chapter 3: The Structure of Web Content – HTML

Transitioning from protocols to content, this chapter introduces HTML as the semantic markup language defining web page structure. It covers document anatomy (DOCTYPE, head, body), metadata elements, and content organization (headings, paragraphs, lists, tables, links, images). Special emphasis is placed on accessibility through proper alt attributes and semantic choices.

User interaction is enabled through HTML forms, with comprehensive coverage of input types, validation attributes, labels, and fieldset organization. HTML5 semantic elements (header, nav, main, section, article, aside, footer) are explained through Document Outline Theory, showing how they create machine-readable structure for SEO, screen readers, and assistive technologies.

Modern layout techniques include responsive images and semantic grids, preparing readers for CSS positioning in Chapter 4. This chapter transforms static markup understanding into structured, interactive document creation.

Learning objectives: Build valid HTML documents; create accessible forms; apply semantic elements for structure. **Applications**: Content management, form processing, accessible web design.

Chapter 4: Design and Layout with CSS

This chapter separates presentation from structure using Cascading Style Sheets. Integration methods (external, internal, inline) and cascade precedence rules are explained, followed by powerful selectors (element, class, ID, combinators, pseudo-classes, attributes) for precise targeting.

The CSS box model (content, padding, border, margin) and measurement units (px, em, rem, %, vw/vh) form the foundation for layout control. Styling techniques cover typography (font families, sizes, weights, line-height), backgrounds, colors, and positioning strategies (normal flow, floats, Flexbox, CSS Grid). Readers learn to create responsive, visually appealing layouts that adapt across devices.

Learning objectives: Control cascade and specificity; master box model and units; build layouts with modern techniques. **Applications**: Responsive design, visual design systems, component styling.

Chapter 5: Introduction to Interactivity with JavaScript

The capstone chapter introduces JavaScript as the language enabling dynamic web applications. It explains JavaScript's role in browsers, integration methods, and basic output techniques, establishing why it is indispensable for modern web development.

Core grammar covers syntax, statements, comments, variables (const, let), data types (string, number, boolean, object), and naming conventions. Operators enable calculations, comparisons, and logical expressions, while control structures (conditionals, loops) and functions support decision-making and code reuse.

Structured data through objects and arrays prepares students for real-world data handling. Events connect JavaScript to user interactions (clicks, form submissions), creating responsive interfaces. The chapter concludes with modern extensions, bridging to advanced topics like APIs and frameworks.

Learning objectives: Write correct JavaScript syntax; handle data and control flow; create interactive elements. **Applications**: Form validation, dynamic content, basic web apps.

Interconnections: Ch. 1 protocols → Ch. 2 HTTP/cookies → Ch. 3 HTML structure → Ch. 4 CSS styling → Ch. 5 JS interactivity. **Practical outcome**: Readers can build complete, interactive websites from foundation to application.

1. Basics of the Internet and the World Wide Web

1.1. Introduction: The Digital Ecosystem

In our modern world, the Internet and the World Wide Web (WWW) have become an integral part of daily life. From information gathering and communication to e-commerce and e-learning, these technologies permeate almost every aspect of our society. For aspiring IT professionals, a sound understanding of the fundamental concepts and architectures that underpin this digital ecosystem is therefore essential. This chapter introduces you to the basic building blocks, explains the difference between the Internet and the WWW, highlights the protocols that enable global communication, and explains the systems for addressing and identifying resources in the vast digital space.

Beyond everyday use, the Internet also forms the backbone of modern digital infrastructure. Governments, businesses, educational institutions, and individuals rely on networked systems to exchange data, deliver services, and collaborate across geographical boundaries. Applications such as cloud computing, streaming platforms, social media, and online collaboration tools all depend on stable and well-structured network architecture. Understanding how these systems interact helps future IT specialists not only to use them effectively but also to design, maintain, and improve them.

A key part of this understanding lies in recognizing the layered architecture that supports Internet communication. Different protocols operate at different levels, each responsible for a specific function such as data transport, routing, or application-level interaction. By examining these layers and the roles they play, students gain insight into how information travels from one device to another across complex global networks. This knowledge

is particularly important when diagnosing network problems, optimizing performance, or ensuring secure communication.

Furthermore, the chapter explores how devices and resources are uniquely identified and located on the Internet. Concepts such as IP addressing, domain names, and resource identifiers allow billions of devices and websites to be organized in a structured and scalable way. Without these systems, navigating the vast amount of information available online would be nearly impossible. Understanding these mechanisms also provides the foundation for topics such as web hosting, domain management, and network configuration.

Finally, the chapter encourages students and practitioners to think critically about the broader implications of Internet technologies. Issues such as security, privacy, reliability, and scalability are central challenges in today's digital society. By developing a conceptual understanding of how the Internet and the Web function, students will be better prepared to analyze these challenges and contribute to the development of future digital solutions. This foundational knowledge will support further learning in areas such as web development, network administration, cybersecurity, and distributed systems.

Internet vs. World Wide Web

Although the terms are often used interchangeably, the distinction is fundamental for us as professionals:

- **The Internet** is the global technical infrastructure – a "network of networks" that physically and logically connects countless computers and networks around the world. It is based on the TCP/IP protocol family, which standardizes data exchange. You can think of it as the global road and rail network for digital data.
- **The World Wide Web (WWW)** is one of many services that are built on top of the Internet. It is a system of linked hypertext documents accessed via the HTTP protocol. Other well-known Internet services

are, for example, e-mail (via SMTP/IMAP), FTP (File Transfer Protocol) or DNS (Domain Name System). The WWW is therefore an application that uses the infrastructure of the Internet – just as a parcel service uses the road network.

The principle of the "Network of Networks"

The Internet can be conceptually imagined as a huge network of interconnected individual networks, (Kurose & Ross, 2020). Data is not transported in one piece, but in small packets from one sender to one recipient. If the sender and recipient are in different networks, these data packets must be forwarded via special switching computers.

Imagine that machine A in network 1 wants to send data to machine E in network 3. The path leads via networks 1, 2 and 3. The data packets are forwarded from A to C, then from C to D and finally from D to E. Machines C and D act as intermediaries here. Depending on the technical layer on which the switching takes place, these devices are referred to as a **bridge**, **router** or **gateway**.

This interconnected structure is the basis for the global reach and robustness of the Internet. But for this communication to work smoothly, a common set of rules is needed. This set of rules forms the technical basis of the Internet: the protocol family.

1.2. The Technological Basis: TCP/IP Protocol Family

The backbone of the Internet is formed by standardized protocols. They are the common language that ensures that devices from different manufacturers and in different places in the world can communicate smoothly with each other. The TCP/IP protocol family (also known as the Internet protocol family) plays a central role here, which defines the rules for data transmission. Two

of the most important protocols on the transport layer are TCP and UDP, which perform opposing but equally important tasks.

Comparing TCP and UDP

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are both used to transport data packets but follow different approaches. Your choice depends crucially on the requirements of the application: reliability or speed?

Table 1. *Comparison between TCP and UDP*

Factor	Transmission Control Protocol (TCP)	User Datagram Protocol (UDP)
Connection Type	Establishes **connection-oriented** communication. A stable connection is established before data is transferred.	Establishes **connectionless** communication. Data is sent without first establishing a connection.
Order	Data packets are sequenced to ensure that they arrive at the recipient in the correct order.	There is no guarantee for the order of the data packets; they can arrive in any order.
Retransmission	Lost packets are detected and automatically re-sent.	Lost data packets are not re-sent. There is no mechanism for recovery.
Guaranteed delivery	The delivery of the data is guaranteed.	Delivery is not guaranteed. Packages can get lost.
Error Checking	Thorough error control (checksums and confirmations) ensures the integrity of the data.	Uses a simple checksum for basic integrity, but without correction mechanisms.
Application example	Web browsing (HTTP/S), email, file transfer (FTP) – where completeness is key.	Video streaming, online gaming, IoT device communication – where speed is more important than 100% reliability.

Key Benefits of the TCP/IP Stack

The TCP/IP protocol family has established itself as a global standard due to several strategic advantages:

- **Scalability & flexibility:** The protocol easily grows with the network and is suitable for small local networks as well as the global Internet.
- **Interoperability & Compatibility:** As an open standard, TCP/IP enables communication between devices and systems from a wide range of manufacturers and prevents monopolies.
- **Reliability & Error Detection:** TCP in particular integrates error detection and correction mechanisms that ensure reliable data transmission for critical applications.
- **Security & Encryption:** The stack is extensible and can be secured by protocols such as TLS (Transport Layer Security) to enable confidential and encrypted communication.

The layer model

To manage the complexity of network communication, the TCP/IP stack is organized into shifts, like a mailroom with different departments. This model is a pragmatic, simplified variant of the theoretical OSI model. Each layer has a specific task and adds its own "packaging" before passing the data packet to the layer below.

From the physical network connection to the application, the Internet Protocol stack consists of the following four layers (from bottom to top):

1. **Device Driver:** Controls the physical transmission of data over the network medium (e.g., Ethernet cable, Wi-Fi signal). It is the hardware level.

2. **IP (Internet layer):** Responsible for addressing and routing data packets through the network. This is where the Internet Protocol (IP) works, giving each packet a sender and destination address.

3. **TCP/UDP (Transport Layer):** Ensures end-to-end communication between the applications on the endpoints. TCP provides reliable transmission, while UDP provides fast but unreliable transmission.

4. **Application:** Contains the protocols for the actual applications we use, such as HTTP for the web, SMTP for emails, or FTP for file transfers.

While TCP/IP defines the rules for data transport, the logical question is: How does a packet know *where* to send in a network of billions of devices? This requires a universal addressing scheme, which we'll look at next.

1.3. Addressing on the Internet: IP Addresses and DNS

For data packets to find their destination on the global Internet, each connected device needs a unique address – comparable to a postal address in the physical world. This addressing system consists of two central pillars: the numerical **IP addresses** for machine processing and the human-readable **domain name**, which make their use practical for us humans.

Function of IP addresses

An Internet Protocol Address (IP) address is a unique numeric identifier assigned to a computer or other device on a network.

- **Uniqueness:** Two machines cannot use the same public IP address on the Internet at the same time.

- **Dynamic assignment:** Internet Service Providers (ISPs) often dynamically assign an IP address to their customers that is only valid for the duration of the online session. This is an efficient way to manage the scarce address pool.

- **Multiple addresses:** A machine can have multiple IP addresses, for example, if it has multiple network interfaces (Wi-Fi, Ethernet).
- **Routing prefix:** For efficient routing, IP addresses are assigned in contiguous blocks. Devices that can be reached via the same path share a common address prefix, which greatly simplifies routing tables on the Internet.
- **Private IP addresses:** Local area networks (such as home or office networks) have special address ranges reserved (such as 192.168.x.y) that are not routed on the public internet.

Hierarchical assignment of IP addresses

The allocation of IP address blocks is strictly hierarchical in order to ensure orderly and decentralized management, (Mockapetris, 1987):

1. **IANA (Internet Assigned Numbers Authority):** The highest authority that manages large blocks of addresses globally.
2. **Regional Internet Registries (RIRs):** IANA delegates blocks to five regional registries, including RIPE NCC (Europe), ARIN (North America), APNIC (Asia-Pacific), LACNIC (Latin America), and AFRINIC (Africa).
3. **Local Internet Registries (LIRs):** The RIRs assign smaller address ranges to LIRs. These are usually Internet Service Providers, who in turn distribute the addresses to their end customers.

Comparison of IPv4 and IPv6

Due to the explosive growth of the Internet – spurred by the rise of mobile devices, streaming services, cloud computing, and billions of connected IoT gadgets – the pool of available addresses in the original Internet Protocol version 4 (IPv4) became critically scarce by the early 2010s.

IPv4 Limitations

IPv4, standardized in 1981, uses 32-bit addresses formatted as four decimal numbers separated by dots (e.g., 192.168.0.1), providing roughly 4.3 billion unique addresses. While sufficient for the early Internet's academic and research networks, this capacity proved inadequate as global adoption surged past 5 billion users by 2026, compounded by address-hungry applications like video conferencing and smart homes.

IPv6 Development

This exhaustion prompted the Internet Engineering Task Force (IETF) to develop IPv6 starting in the mid-1990s as a successor protocol. IPv6 employs 128-bit addresses, written in hexadecimal with colons (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334), yielding approximately 340 undecillion (3.4 × 10^38) addresses – virtually inexhaustible even for future planetary-scale networking.

Key Improvements and Adoption

Beyond sheer capacity, IPv6 eliminates the need for Network Address Translation (NAT), enabling true end-to-end connectivity, simplifies packet headers for faster routing, includes built-in IPsec security, and supports auto-configuration via stateless address autoconfiguration (SLAAC). Though rollout began in 1998 with RFC 2460, full global adoption lags at around 40–50% as of 2026 due to compatibility challenges, but dual-stack implementations (running IPv4 and IPv6 side-by-side) ensure a smooth transition.

Table 2. *Comparison IPv4 vs. IPv6*

Feature	IPv4	IPv6
Address length	32-bit	128-bit
Address space	Approx. 4.3 billion addresses	Virtually unlimited (3.4 x 10^{38} addresses)

Feature	IPv4	IPv6
Presentation	Decimal, separated by dots (e.g. 192.0.2.1)	Hexadecimal, separated by colons (e.g. 2001:0db8::8a2e:0370:7334)
Pros	Widely used and established	**Enormous address space** for the "Internet of Things" (IoT), integrated security functions (IPsec), more efficient routing through simpler headers

The role of the Domain Name System (DNS)

Since numeric IP addresses are difficult for humans to remember, the Domain Name System (DNS) was introduced, (Mockapetris, 1987). In didactically simplified terms, it can be imagined as the **"telephone book of the Internet"**: It translates human-readable domain names such as orf.at into the associated machine-readable IP addresses.

DNS query flow

When a user wants to access a web page, a standardized process runs in the background that uses the different layers of the TCP/IP model:

1. **DNS request:** The user enters a URL (such as www.campus02.at) into the browser. This triggers a DNS request at the **application layer**. The operating system sends this request (wrapped in UDP/TCP and IP packets) to a DNS server to obtain the associated IP address.

2. **DNS response:** The DNS server responds with the numeric IP address of the web server. This response also goes through the TCP/IP stack back to the client.

3. **HTTP communication:** The browser uses the IP address obtained to establish a new TCP connection to the web server and request the actual website via the HTTP protocol (again at the **application layer**).

DNS hierarchy and other features

The DNS is hierarchically structured like a tree, which enables decentralized and scalable management. A domain name is read from right to left:

Example: c02-NB-061.campus02.at

- **. (Root):** The (often invisible) root of the DNA tree.
- **at:** The top-level domain (TLD).
- **campus02:** The second-level domain that is registered by an organization.
- **c02-NB-061:** The hostname that identifies a specific device within the domain.

In addition to pure name resolution, DNS offers other functions, such as the definition of alias names (CNAME records), which allow one hostname to refer to another (e.g. shop.example.com refers to www.example.com).

Now that it's clear how devices are found on the web, let's look at the user's primary tool for accessing the content of the web: the browser.

1.4. The Gateway to the Web: Browsers and Standards

The web browser is the central web client and the primary tool for users to access and interact with the content of the World Wide Web. Its main task is to retrieve HTML, CSS and JavaScript code from a web server, interpret it and present it as a visually prepared, interactive website, (Mozilla Developer Network [MDN], 2025).

Core features of web browsers

Modern browsers offer a variety of features that make web browsing easier and personalized:

- **Tabs:** Allow you to browse multiple web pages in parallel in one window.
- **Address bar:** Used to enter URLs and search.
- **Navigation:** Buttons to scroll forward and backward in the visit history.
- **Refresh/Reload:** Reloads the currently displayed page from the server.
- **Home Button:** Leads to the predefined start page.
- **Bookmarks:** Used to store favorite web pages for quick access.
- **Download history:** Shows an overview of the downloaded files.
- **Extensions/Add-ons:** Allow you to add additional, specialized features to the browser.

Market shares of browsers

The browser market is dominated by a few large providers. According to data from Statcounter for July 2025, **Chrome** is the undisputed leader with a market share of **67.94%.** It is followed by **Safari** (**16.18%), Edge** (**5.07%)** and **Firefox** (**2.45%).** This dominance has significant implications for web development, as developers need to ensure that their applications work flawlessly on the most used browsers.

The importance of standardization through the W3C

To ensure that the web works consistently across browsers and devices, open standards are essential. Without them, we would fall back into the chaos of the early "browser wars", in which websites had to be developed separately for each browser. The most important international organization for the standardization of web technologies is the **World Wide Web Consortium**

(W3C). It develops technical specifications and guidelines to ensure the long-term growth, interoperability and accessibility of the web.

The W3C standardization process

At the W3C, a standard goes through several stages of maturity before it is officially adopted. The final stages of this process are:

- **Proposed Recommendation (PR):** A finalized draft that is submitted to the W3C Advisory Council for final vote.
- **W3C Recommendation (REC):** Once approved, the document becomes an official W3C standard that serves as a stable foundation for developers and browser manufacturers.

Well-known W3C standards include **HTML, XML, CSS**, **PNG**, **SVG** and the **DOM** (Document Object Model). One of these basic standards specifies how resources are uniquely addressed on the web, which leads us to the next topic.

1.5. Identification of Resources: URI, URL and URN

In order to be able to access a certain piece of information on the huge World Wide Web, it must be uniquely identifiable. The universal system for this is the Uniform Resource Identifier (URI). A "resource" is a very broad term and can be anything: a website, an image, a video, but also a person, a company or even an abstract concept such as a mathematical operator.

Differentiation: URI, URL, and URN

URI is the generic term that can be divided into two specialized types: URL and URN. Every URL and every URN is also a URI, but not the other way around.

Table 3. *Definitions URI, URL and URN*

Type	Purpose	Example
URI (Uniform Resource Identifier)	Generic term used to identify a resource by name, location, or both.	https://example.com or urn:isbn:3772305989
URL (Uniform Resource Locator)	Specifies the **location** of a resource and describes how it can be accessed (localization).	https://www.campus02.at/index.html
URN (Uniform Resource Name)	Specifies a unique and persistent **name** for a resource, regardless of its current location.	urn:isbn:3772305989 (uniquely identifies a book by its ISBN)

In daily use, the URL is by far the most common form of URI that we encounter on the web.

Structure of a URI

A generic URI can consist of up to five main components, separated by specific characters.

Example: foo://example.com:8042/over/there?name=ferret#nose

1. **Scheme** (foo) Specifies the protocol or schema used to access the resource (e.g., http, https, ftp, mailto). It defines the "language" with which it is spoken.
2. **Authority** (example.com:8042) Identifies the authority (usually a server) that manages the resource. It consists of:
   - [userinfo@] (optional): Login information such as user:password. Hardly used today for safety reasons.
   - host: The host name (e.g. example.com) or IP address (IPv4: 192.168.5.2, IPv6: [2001:db8::1]).
   - [:p location] (optional): The port number. If it is omitted, the default port of the schema is used (e.g. **80** for HTTP, **443** for HTTPS).

3. **Path** (/over/there) Defines the hierarchical path to the resource on the host, similar to a file system structure. The segments are separated by forward slashes (/).

4. **Query** (?name=ferret) Starts with a question mark (?) and contains non-hierarchical data that is passed to the resource. Often these are key-value pairs (key=value) that are separated by & and transmit, for example, filter or search parameters.

5. **Fragment** (#nose) Starts with a pound sign (#) and identifies a secondary part *within* the resource. This part is processed client-side (in the browser) and is **not** sent to the server.
   - **HTML:** Jump to an element with a specific id (e.g. ...#C15).
   - **PDF:** Display of a specific page (e.g. ...#page=23).
   - **Video/Audio:** Jump to a specific time indicator (e.g., ...#t=65 for 65 seconds).

URL Shortener

URL shorteners are services that convert long and complex URLs into short, easy-to-share alias links. Technically, this works via an HTTP redirect, usually with the status code 301 Moved Permanently.

Table 4. *Pros and Cons of URL shortening*

Pros	Cons
Shorter, more readable links, ideal for social media and print.	**Obfuscated target:** The original target is not visible, which can be abused for phishing and malware.
Possibility to assign your own, memorable alias names.	**Dependency:** The short link only works as long as the provider of the service exists and is reachable.
Space saving in posts with character limits (e.g. Twitter).	**Vulnerabilities:** Can be used for attacks such as cross-site request forgery (CSRF).

The fundamentals covered in this chapter – from building the Internet to protocols and addressing and identifying resources – are the foundation for understanding more specific technologies such as the HTTP protocol, HTTP cookies, and caching strategies, which will be covered in the next chapter.

1.6. Security Foundations: HTTPS and TLS

While HTTP enables communication, HTTPS (HTTP over TLS) is essential for secure data transfer, encrypting traffic to prevent eavesdropping and tampering. TLS 1.3 (standard since 2020) uses perfect forward secrecy and eliminates weak ciphers, with 95% global adoption by 2026 per Google Transparency Report. Insert code example: <meta http-equiv="Content-Security-Policy" content="default-src 'self'"> for CSP basics, tying into browser standards. This prepares readers for real-world deployment, where mixed content warnings block insecure sites.

2. Communication and Performance

2.1. The Hypertext Transfer Protocol (HTTP)

The **Hypertext Transfer Protocol (HTTP)** is the fundamental protocol used for the transmission of data on the **World Wide Web**. It defines how messages are formatted and transmitted between different systems on the internet, enabling web browsers, servers, and other applications to communicate with each other. Whenever a user accesses a website, downloads a file, or submits information through a web form, HTTP is the protocol responsible for carrying these requests and responses between the client and the server.

HTTP operates according to a **client–server model**. In this model, a client – most commonly a web browser – initiates communication by sending a request to a web server. The server processes the request and returns a response, which may include HTML documents, images, videos, or other types of resources. This request–response mechanism forms the basis of nearly all interactions on the web. For example, when a user enters a URL into a browser, the browser sends an HTTP request to the corresponding server, which then responds by delivering the requested web page.

One important characteristic of HTTP is that it is a **stateless protocol**. This means that each request from a client to a server is treated as an independent transaction. The server does not automatically remember previous interactions with the same client. While this simplifies the design and scalability of web systems, it also means that additional mechanisms are required to maintain user sessions or persistent data. Technologies such as cookies, sessions, and tokens are therefore often used alongside HTTP to enable features like user authentication, shopping carts, and personalized content.

Over time, HTTP has evolved to support the growing complexity and performance requirements of the modern web. Newer versions such as HTTP/2 and HTTP/3

introduce improvements like faster data transfer, multiplexing, and better resource management. Despite these advancements, the core principle of HTTP remains the same: enabling reliable and standardized communication between clients and servers across the internet.

Understanding how HTTP works is essential for anyone studying web technologies, networking, or software development, as it forms the backbone of how information is requested, transmitted, and displayed on the web.

Important HTTP Methods

HTTP defines a set of request methods ("verbs") that specify what action to take with the requested resource, (Fielding et al., 1999)..

Table 5. *Most important HTTP methods*

Method	Purpose and side effects	Typical Achievement Status Codes
GET	Requests the specified resource from the server. This method should be safe and without side effects, i.e. it does not change the state on the server.	200 (OK)
PUT	Creates a new resource under the specified URI or completely replaces an existing one. PUT is **idempotent**: Executing the same request multiple times has the same effect as executing it once.	201 (Created) for new creation, 200 (OK) or 204 (No Content) for change
DELETE	Requests the deletion of the specified resource on the server.	200 (OK), 202 (Accepted), 204 (No Content)

HTTP header

HTTP messages, both requests and responses, contain metadata in the form of headers. These provide important information about the message itself or its content.

Table 6. *HTTP Headers*

Request Header	Response Header
Accept-...: Specifies which content types (e.g., text/html) or languages the client can process.	**Content-Type**: Specifies the media type of the resource in the response body (e.g. text/html).
	Content-Length: Specifies the size of the response body, in bytes.
	Set-Cookie: Instructs the client to store a cookie to simulate state across multiple requests.

Note: The **status code** (e.g. 200 OK, 404 Not Found) is technically not a header, but the most important part of the first line of the server response (the status line) that indicates the result of the request.

The efficiency of this HTTP communication can be increased considerably. One of the most important techniques for this is caching, which we will now take a closer look at.

2.2. State Management with HTTP Cookies

The HTTP protocol is **stateless** in nature. This means that every request is isolated from the server and handled without knowledge of previous requests. However, for many web applications, such as online stores (shopping cart) or personalized portals (login status), it is essential to maintain a user's state across multiple page views. HTTP cookies are the classic client-side solution to this problem and form the basis for personalized web experiences.

2.2.1. Definition and Operation of Cookies

A **cookie** is a small piece of text information that is stored in the user's browser by a website. It is either sent directly from the web server or generated by a client-side script (e.g. JavaScript).

The functionality is based on two HTTP headers:

1. **Setting the cookie**: In its HTTP response, the server sends the set cookie header, which instructs the browser to store a cookie with a specific name, value, and optional attributes.
2. **Sending the cookie**: For each subsequent request to the same domain, the browser automatically sends all stored, valid cookies in the cookie header. The server can read this information and recognize the user.

Primary uses of cookies:

- **Session management**: maintain login status, save the shopping cart.
- **User identification/personalization**: Remembering preferences such as language, design, or other personalized content.
- **Tracking**: Analysis of user behavior across one or more websites, often for advertising purposes.

2.2.2. Anatomy of a cookie: attributes and their meaning

A cookie consists not only of a name and a value, but also of several attributes that control its behavior, (Barth, 2011).

- **Domain**: Determines which domain the cookie is valid for.
- **Path**: Restricts the validity of the cookie to a specific path within the domain.
- **Expires**: Defines an exact expiration date and time. After this time, the cookie will be deleted.
- **Max-Age**: Defines the lifetime of the cookie in seconds from the time it is created.

Security-Relevant Attributes

These attributes are crucial to protect cookies from common web attacks.

- **HttpOnly**: Prevents the cookie from being read via client-side scripts (JavaScript). This is a critical measure against **Cross-Site Scripting (XSS).** Without HttpOnly, an attacker could inject malicious JavaScript code using an XSS vulnerability. This code could execute document.cookie in the victim's browser and exfiltrate the session cookie to the attacker's server, which could lead to a complete takeover of the user account.

- **Secure**: Ensures that the cookie is only sent over an encrypted HTTPS connection. This protects against eavesdropping on cookie data on unsecured networks (e.g. public Wi-Fi).

- **SameSite**: Checks whether a cookie is sent with requests from third-party domains (cross-site). This is the primary defense mechanism against **Cross-Site Request Forgery (CSRF)** attacks, in which an attacker tricks a logged-in user into unknowingly performing an action on another website (such as a transfer in a banking app). It has three possible values: Strict, Lax (default in modern browsers), and None (requires the Secure flag).

Example of a secure set cookie header:

Set-Cookie: SESSION_ID=abc123; HttpOnly; Secure; SameSite=Strict; Path=/; Max-Age=3600

• *SESSION_ID=abc123: name and value of the cookie.*

• *HttpOnly: Protection against XSS.*

• *Secure: Transmission only via HTTPS.*

• *SameSite=Strict: Protection against CSRF.*

• *Path=/: Valid for the entire domain.*

• *Max-Age=3600: Valid for one hour (3600 seconds).*

2.2.3. Classification of Cookie Types

Cookies can be classified according to several characteristics, most commonly **their lifespan, their origin, and their purpose**. These categories help to better understand how cookies function and how they are used by websites to manage user interactions, store information, and support web services.

Classification by Lifespan

Cookies can be divided into **session cookies** and **persistent cookies** based on how long they remain stored on the user's device.

- **Session cookies** are temporary cookies that exist only for the duration of a user's browsing session. They are automatically deleted when the browser is closed. These cookies are commonly used to maintain temporary information, such as keeping a user logged in while navigating between pages or remembering items in a shopping cart during a single visit.
- **Persistent cookies** (also called **permanent cookies**) remain stored on the user's device for a predefined period of time, which can range from a few days to several months or even years. They allow websites to recognize returning users, remember preferences such as language settings, and provide a more personalized browsing experience.

Classification by Origin

Cookies can also be categorized according to where they originate from.

- **First-party cookies** are created and stored by the website that the user is currently visiting. They are typically used to support core website functions such as authentication, user preferences, and session management.
- **Third-party cookies** are created by domains other than the one the user is visiting.

◦ **Third-party cookie:** Is set by a different domain than the one the user is currently visiting (for example through embedded advertising banners, social media plugins, or external analytics services). These cookies are mainly used for **cross-site tracking**, allowing companies to follow users across multiple websites in order to analyze behavior or display targeted advertising. Due to privacy concerns, third-party cookies are increasingly **restricted or blocked by modern web browsers**.

Classification by Purpose

Another important way to classify cookies is by their function or intended use.

- **Essential (or strictly necessary) cookies** are required for the basic operation of a website. Without them, certain services such as login functionality, secure areas, or shopping carts would not work properly.
- **Functional cookies** store user preferences and settings to improve usability, such as remembering language choices, region settings, or display preferences.
- **Analytics or performance cookies** collect information about how users interact with a website. This data helps website operators understand visitor behavior, identify technical problems, and improve the overall user experience.
- **Marketing or advertising cookies** are used to track users across websites in order to display targeted advertisements that are relevant to their interests.

Special Forms: Tracking and Zombie Cookies

Some cookies represent **special or problematic implementations** that raise significant privacy concerns.

- **Tracking cookie:** Used to analyze a user's browsing behavior across different websites in order to create detailed **user profiles**. These

cookies often collect data about visited pages, clicks, and time spent on websites. Most tracking cookies are **third-party cookies** and are commonly used by advertising networks and analytics providers.

- **Zombie cookie:** A cookie that, once deleted by the user, **automatically recreates itself**. This is achieved by storing the cookie data in multiple locations (for example in browser storage or cached files) and restoring it if the original cookie is removed. This practice bypasses user control and is widely considered a **serious violation of user privacy**.

By classifying cookies according to **lifespan, origin, purpose, and special forms**, it becomes easier to understand both their **technical role in web applications** and their **implications for privacy and data protection**.

2.2.4. Data Protection, GDPR and User Rghts

The use of cookies is subject to strict legal regulations, in particular the **GDPR (General Data Protection Regulation)** and the **ePrivacy Directive**.

- **Technically necessary cookies**: Serve the basic functionality of a website (e.g. shopping cart cookie, session login cookie). They may be set without the explicit consent of the user.
- **Non-necessary cookies**: All other cookies, in particular for marketing, statistics and tracking. Active **and informed consent (opt-in)** of the user is required for their use. A pre-ticked box in a cookie banner is not allowed.

Users have extensive rights in the handling of their data:

- **Right to consent**: The right to accept or reject cookies.
- **Right to revocation**: It must be possible to easily revoke consent at any time.

- **Right to information**: Users must be informed about which cookies are stored and for what purpose.
- **Right to erasure**: Users can have their data and the cookies associated with it deleted.

2.2.5. Alternatives to Cookies

For client-side data storage, there are modern alternatives that can be more beneficial in certain scenarios.

- **Web Storage (LocalStorage and SessionStorage):** Provides significantly more storage space than cookies (5-10 MB). The data is not automatically sent to the server with every HTTP request, which reduces overhead.
  - ***LocalStorage*** stores data permanently until it is manually deleted.
  - ***SessionStorage*** only stores data for the duration of a browser session (until the tab is closed).
- **IndexedDB:** A full-fledged client-side database in the browser designed for storing large and complex amounts of data, as well as for offline applications.

Table 7. *Comparison of storage mechanisms*

Criteria	Local Storage	Session Storage	Cookies
Storage Capacity	5-10 MB	5-10 MB	4 KB
Auto Expiry	No	Yes (on tab close)	Yes (configurable)
Server Side Accessibility	No	No	Yes
Data Transfer HTTP Request	No	No	Yes
Data Persistence	Till manually deleted	Till browser tab is closed	As per expiry TTL set

While cookies remain essential for server-side state management, client-side storage with Web Storage has the advantage of eliminating HTTP request overhead. Every unnecessary byte transfer, such as the automatic sending of cookies, is a performance burden. The systematic reduction of such latencies is the core goal of caching strategies, which we will address in the next chapter.

2.3. Web Caching to Increase Performance

Caching is a fundamental technique used to improve the **performance and efficiency of web applications**. The main idea behind caching is to store temporary copies of frequently requested resources so that subsequent requests can be served faster. Instead of retrieving the same data repeatedly from the origin server, the system can deliver the cached version, which significantly reduces the time required to respond to user requests.

The benefits of caching are substantial. By serving cached content, **latency for the user decreases**, because data can be delivered more quickly. At the same time, the **load on the server infrastructure is reduced**, since fewer requests need to be processed by the backend systems. Additionally, caching helps to **save network bandwidth**, as the same resources do not need to be transferred repeatedly over the network. For modern high-traffic websites and applications, caching is therefore an essential optimization strategy.

Caching Types and Strategies

Caching can be implemented in different ways depending on where the cached data is stored and how it is managed. Various **cache types and strategies** are used to balance performance, storage capacity, and scalability.

Types of Caches

- **In-memory caches**
  In-memory caches store data directly in the system's **random access**

memory (RAM). Because memory access is extremely fast, these caches provide very low latency and are ideal for frequently accessed data such as session information or database query results. Typical technologies used for in-memory caching include Redis and Memcached.

- **Disk-based caches**
  Disk-based caches store cached data on a storage device such as a hard drive or SSD. While this approach is slower than in-memory caching, it allows significantly larger amounts of data to be stored. Disk-based caches are commonly used for web content such as static files, images, or previously requested pages. Examples include Varnish Cache, Squid, and the cache built into modern web browsers.

- **Distributed caches**
  In distributed caching systems, the cache is spread across **multiple servers or nodes** within a network. This architecture allows the system to scale horizontally and provides increased fault tolerance. Distributed caching is often used in large cloud-based systems and high-traffic applications. Examples include Amazon ElastiCache and Apache Ignite.

Cache Invalidation

Although caching improves performance, it introduces the challenge of **keeping cached data up to date**. If outdated data remains in the cache, users may receive incorrect or stale information. Therefore, systems must implement mechanisms for **cache invalidation**, which means updating or removing outdated entries.

Common invalidation strategies include:

- **Time-based invalidation**
  Cached entries automatically expire after a predefined period called **Time-To-Live (TTL)**. Once the TTL expires, the cached item is removed or refreshed.

- **Event-based invalidation**
  The cache is updated whenever the underlying data changes. For example, when a database record is modified, the related cached entry is automatically refreshed or deleted.

- **Manual invalidation**
  In some cases, administrators or application logic explicitly clear or refresh cache entries when necessary.

- **HTTP method invalidation**
  In web systems, certain HTTP methods imply that a resource has changed. For instance, requests using **POST, PUT, or DELETE** usually invalidate the cached response of a previous **GET request**, because the resource is assumed to have been modified.

Caching controls in HTTP

The **HTTP protocol** provides several mechanisms that allow clients and servers to manage caching behavior precisely. These mechanisms rely mainly on special HTTP headers that control validation and freshness of cached resources.

Last-Modified / If-Modified-Since

This mechanism allows a client to check whether a resource has changed since it was last retrieved.

1. When the resource is requested for the first time, the server sends the **Last-Modified** header containing the timestamp of the resource's last modification.

2. During subsequent requests, the client sends this timestamp back to the server in the **If-Modified-Since** header.

3. The server compares the timestamps. If the resource has not changed, it responds with the status code **304 Not Modified** and no message body.

Because the client already has a valid cached version, it can simply reuse the local copy instead of downloading the resource again.

E-Tags

Entity Tags (E-Tags) provide a more precise and reliable validation mechanism than modification timestamps. An E-Tag is a **unique identifier**, often generated as a hash value, representing a specific version of a resource.

Typical workflow:

1. **First request**
   The client requests a resource. The server responds with **HTTP 200 OK**, the resource itself, and an **E-Tag header** (for example ETag: "hb852-dfm55-8khuk"). The client stores both the resource and the E-Tag.

2. **Subsequent request**
   When the client requests the same resource again, it includes the stored identifier in the **If-None-Match** header.

3. **Server validation**
   The server compares the received E-Tag with the current identifier of the resource.
   - If the values match, the server responds with **HTTP 304 Not Modified**, allowing the client to reuse its cached copy.
   - If they differ, the server returns **HTTP 200 OK** along with the updated resource and a new E-Tag.

Advantages of e-tags

E-Tags provide several important benefits:

- **Performance:** Reduces bandwidth and server load by avoiding unnecessary data transfers.

- **Data consistency:** Ensures that the client is always using the most up-to-date version of a resource, even if the last-modified date does not change.
- **Synchronization:** Helps to prevent conflicts during simultaneous edits ("optimistic locking").
- **Server-side control:** The server has full control over the criteria for generating and managing the e-tags.

Challenges with e-tags

In distributed architectures where multiple servers operate behind a **load balancer**, E-Tags may create challenges. If different servers generate slightly different identifiers for the same resource, clients may incorrectly assume that the resource has changed. This leads to unnecessary reloads and reduced caching efficiency. To avoid this problem, systems must ensure **consistent E-Tag generation across all servers**.

Service Worker

A modern and highly powerful caching technology is the Service Worker. A service worker is a JavaScript program executed by the browser in the background. It acts as a programmable proxy between the web application and the network, intercepting requests and allowing developers to implement advanced caching strategies.

With service workers, developers can cache resources dynamically, control network requests, and even provide complete offline functionality. This capability forms the technological foundation for Progressive Web Apps (PWAs), enabling web applications to behave similarly to native mobile apps

2.4. Next-Gen Protocols: HTTP/2, HTTP/3, and QUIC

HTTP/2 introduces multiplexing, header compression, and server push for 20–50% faster loads; HTTP/3 over QUIC (UDP-based) reduces connection times by 30% via 0-RTT handshakes. QUIC integrates TLS 1.3 natively, mitigating head-of-line blocking. Example browser check: navigator.userAgent.includes ('HTTP/3'). As PWAs and CDNs dominate, these protocols are now default in Chrome/Edge, enhancing caching efficacy discussed earlier

After exploring the communication protocols and performance optimization techniques that support modern web applications, we can now turn our attention to the language that defines the **structure and content of web pages: HTML**.

3. The Structure of Web Content – HTML

3.1. Introduction to HTML

The **HyperText Markup Language (HTML)** is the standard markup language for creating and structuring content on the World Wide Web. HTML forms the semantic framework of a website – it defines the meaning and structure of content such as headings, paragraphs, images and links. It is the basic layer for the content on which CSS (for design and layout) and JavaScript (for interaction) are built.

HTML's semantic approach improves SEO, accessibility (via ARIA attributes), and maintainability, forming the skeleton that CSS styles into visually appealing layouts and JavaScript animates for user engagement. Without a solid HTML foundation, advanced frameworks like React or Vue.js cannot function effectively.

Anatomy of an HTML element

An HTML document consists of a hierarchy of elements. Most elements consist of three parts:

1. **Start tag:** A name enclosed in angle brackets (for example,).
2. **Content:** The actual text or other HTML elements nested within it.
3. **End tag:** Same name as in the start tag, but preceded by a forward slash (for example,).

Empty Elements

Some HTML elements have no content and therefore do not need an end tag. They are referred to as empty or contentless elements. A typical example is the
 tag, which inserts a simple line break, or the <img> tag for images.

Block vs. Inline Elements

HTML elements can be divided into two basic display categories:

- **Block elements:** These elements always start on a new line and by default take up the entire available width of their container. They are used to structure larger content blocks.
    - Examples: <p> (paragraph), <h1> to <h6> (headings), (a generic container).
- **Inline elements:** These elements do not create a line break and only take up as much width as their content needs. They are used to mark up sections of text *within* a block element.
    - Examples: (important text), (a generic inline container), <code> (code snippet).

Attributes

Attributes provide additional information about an HTML element and are always specified in the start tag. They consist of a name and a value in the form name="VALUE". Some attributes are required for certain tags, while others are optional.

Table 8. *HTML Attributes*

Attribute	Typical element	Description
href	<a>	Specifies the destination URL of a link.
src	<img>	Specifies the source (URL) of an image or other resource.
old	<img>	Defines an alternative text for an image (crucial for accessibility).
Style	All Elements	Used to specify specific inline CSS styles.

Attribute	Typical element	Description
long	<html>	Defines the language of the document.
title	All Elements	Displays additional information as a tooltip when hovering over it.
width, height	<img>	Defines the width and height of an image.

These individual building blocks are assembled into a complete HTML document that has a clearly defined basic structure.

3.2. The Basic HTML Document Structure

Every HTML document follows a basic structure that tells the browser how to interpret the page, (World Wide Web Consortium [W3C], 2024).

```
<! DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>page title</title>
</head>
<body>
   <h1>This is a heading</h1>
   <p>This is a paragraph.</p>
</body>
</html>
```

- **<! DOCTYPE html>**: This declaration always comes first and defines the document type. It tells the browser that it is an HTML5 document.

- **<html>**: This is the root element that wraps the entire HTML page. The lang attribute specifies the primary language of the document.
- **<head>**: This area contains metadata about the document that is not directly visible on the page. This includes the title, character set information or links to external files.
- **<body>**: This area contains all the visible content of the website, such as text, images, links and tables.

Elements in the <head> area

The **<head>** container is crucial for the correct functioning, display and findability of a website.

- **<title>**: Defines the title of the document. This title is visible in three important places:
  1. In the title or tab bar of the browser.
  2. As a name for bookmarks/favorites.
  3. In the results of search engines (SEO-relevant).
- **<meta>**: Used to specify metadata such as the character set (e.g. <meta charset="UTF-8">), the page description for search engines, keywords or the author.
- **<style>**: Used to embed internal CSS styles directly into the HTML document.
- **<link>**: Links the HTML document to external resources. The most common application is to include an external CSS file or **favicon** (the small icon in the browser tab).
  - Example of a favicon: <link rel="icon" type="image/x-icon" href= "/images/favicon.ico">

- **<script>**: Used to embed client-side JavaScript code or link to an external JavaScript file.
- **<base>**: Defines a base URL for all relative URLs in the document. This can be useful, but should be used with caution as it affects all relative paths.

From the invisible but essential <head> area, we now come to the visible content, which is structured in the <body>.

3.3. Structuring Content in the <body>

The semantically correct structuring of content in the **<body>** tag is more than just a technical exercise. It is the basis for search engine optimization (SEO), accessibility and the general readability of the code. Using the right HTML tag for each content is not just a goal in itself, but a central principle: we create a machine-readable document structure that benefits search engines, assistive technologies (such as screen readers) and other developers alike.

Headings and paragraphs

- **Headings (<h1> – <h6>):** HTML provides six levels of headings to structure content hierarchically. <h1> should be used for the main heading of the page and ideally only occur once per page. <h2>, <h3> etc. are used for subheadings in logical order.
- **Paragraphs (<p>):** The <p> tag is used to define blocks of text as paragraphs. Browsers automatically add vertical spacing before and after a paragraph.

Links (hyperlinks)

Links created with the <a> (Anchor) tag are at the heart of hypertext.

- The href attribute specifies the destination of the link.
- The target attribute determines where the link is opened (e.g. target="_blank" opens the link in a new tab).
- **Image as link**: An image can be turned into a clickable link by placing the **<img>** tag inside a **<a>** tag.
- **Email link**: The mailto: scheme in the href attribute opens the user's email program: **… .**
- **Bookmarks**: You can link to specific sections within the same page. To do this, an id is assigned to the target element (e.g. <h2 id="C15"> Chapter 15</h2>) and the link points to this ID with a fragment identifier (Go to Chapter 15).

Images

Images are inserted with the empty inline element **<img>**.

- **src**: This required attribute specifies the path (URL) to the image.
- **alt**: This attribute contains an alternative text. A crucial point that you should never neglect in your practice is the alt attribute. It's not just a fallback in case the image fails to load; it is the bridge to accessibility for screen reader users and a key signal for search engines to understand the image content.
- **width** and **height**: Define the dimensions of the image, which helps the browser reserve the space when the page loads.
- **Background image**: A background image is not set with HTML, but with CSS, typically via the style attribute and the property background-image: .
- **Responsive images**: The <picture> element makes it possible to provide different versions of images for different screen sizes or resolutions.

It contains several <source> elements from which the browser chooses the most suitable one to save bandwidth and optimize the display.

Lists

- **Unordered Lists (<ul>):** Create a bulleted list where each item (<left>) is marked with a bullet point. The CSS property list-style-type can be used to change the character (disc, circle, square, none).
- **Ordered Lists (<ol>):** Create a numbered list where each item (<left>) is numbered consecutively. The type attribute controls the numbering format.

Table 9. *Lists in HTML*

type attribute	Result
"1" (default)	Numbers (1, 2, 3, ...)
"A"	Capital letters (A, B, C, ...)
"a"	Lowercase letters (a, b, c, ...)
"I"	Roman numerals, large (I, II, III, ...)
"i"	Roman numerals, small (i, ii, iii, ...)

Tables

Tables are defined with **<table>** and are used for the structured representation of tabular data in rows (**<tr>**) and cells (**<td>**).

- For column headers, the **<thead>** section uses the **<th>** (table header) tag instead of **<td>** (table data). This makes the header stand out semantically and often visually.
- The colspan and rowspan attributes can be used to connect cells across multiple columns or rows.

Text formatting and special characters

HTML provides tags for semantic markup of text. It's important to distinguish between semantic tags (which convey meaning) and purely presentational tags.

Table 10. *Text formatting and special characters tags in HTML*

Day	Description	Semantics vs. Presentation
	Strongly important text	Semantic (important)
<b>	Bold text	Presentation (visually bold)
	Highlighted text	Semantic (emphasized)
<i>	Italic text	Presentation (visual italics)
<sub>	Subscript text	Semantic
<sup>	Superscript text	Semantic
<del>	Deleted text	Semantic
<ins>	Inserted text	Semantic
<mark>	Highlighted/highlighted text	Semantic

Characters that have a special meaning in HTML (such as < and >) or are not available on the keyboard must use entities.

Table 11. *Entities and Unicode notations for special characters*

Signs	Entity Notation	Unicode Notation
&	&	&
<	<	<
>	>	>
"	"	"
ü	ü	ü
Ü	Ü	Ü

In addition to the pure presentation of content, HTML also offers special elements for capturing user input: forms.

3.4. Interacting with Users: HTML Forms

HTML forms are the central element for interacting with users. They are used to collect data that can then be sent to a server for processing, such as when registering, logging in, placing an order, or making a search query.

The <form> element

The <form> element serves as a container for all form elements. Its attributes control how and where the data is sent.

Table 12. *Attributes for form HTML-Tag*

Attribute	Description
Action	Specifies the URL of the server-side script that is to process the form data.
method	Specifies the HTTP method to use (typically GET for queries or POST for data delivery).
enctype	Defines the encoding of the form data. Important for file uploads: "multipart/form-data".
novalidate	Disables default browser validation for the entire form when set.

Form elements in detail

- **<input>**: The most versatile form element. The type is defined by the type attribute (e.g. text, password, radio, checkbox, submit, file, date, color).
- **<label>**: Defines a label for an input element. This is crucial for accessibility and usability. The link is made via the label's for attribute, which should refer to the id of the associated input element.

- **<select> and <option>**: Creates a drop-down list. <select> is the container, and each <option> element represents a selectable entry. The multiple attribute can be used to allow multiple selections, and selected can be used to preselect an option.
- **<textarea>**: Defines a multi-line input field for longer texts. The visible size can be controlled via the rows and cols attributes.
- **<button>**: Represents a clickable button that can be used to submit the form (type="submit") or to trigger JavaScript actions (type="button").
- **<fieldset> and <legend>**: Used to visually and semantically group related form fields. <fieldset> encloses the group, and <legend> provides the appropriate heading for that group.
- **<datalist>**: Provides a list of predefined options as suggestions for a <input> element as the user types, improving the user experience.

HTML5 has introduced new semantic elements beyond forms, which have revolutionized the structuring of entire page layouts.

3.5. Modern Page Layout with HTML5

Before HTML5, page layouts were often built with a flood of non-specific elements and id or class attributes (** … **). HTML5 has introduced a set of semantic layout tags that promote a clearer separation of structure and representation, World Wide Web Consortium [W3C], 2024). These tags make the code more understandable to developers, browsers, and search engines because they describe the meaning of the different areas of a web page directly in the markup.

HTML5 Semantic Layout Tags

- **<header>**: Defines the header area of a page or section. Typically includes the logo, page title, and main navigation.

- **<nav>**: Wraps around the main navigation area of the page with the most important links.
- **<section>**: Group thematically related content, such as a chapter or a specific topic block on a landing page.
- **<article>**: Represents a self-contained, self-contained piece of content that could be distributed independently of the rest of the page (e.g., a blog post, a news article).
- **<aside>**: Contains content that is only marginally related to the main content, such as a sidebar with further links, glossary terms or advertisements.
- **<footer>**: Defines the footer of a page or section. Typically contains copyright information, contact details and links to the imprint or data protection.

Responsive design

Responsive web design is a design approach that flexibly designs the layout of a web page to provide optimal user experience on a variety of devices, from large desktop monitors to tablets and small smartphone displays. This is achieved through a combination of flexible grids, adaptable images, and CSS media queries, (Powell, 2010).. Elements are automatically rearranged, zoomed in, shrunk, or hidden based on the screen size to ensure a consistent and accessible experience.

3.6. Semantic HTML5 Elements – Structure Theory

3.6.1. Why Semantics Matter: From Presentation to Meaning

Traditional HTML used and everywhere, creating "div soup" with no inherent meaning. Semantic HTML5 elements express **content purpose**, helping:

- **Screen readers** (accessibility)
- **Search engines** (SEO)
- **Developers** (self-documenting code)
- **Future CSS/JS** (targeting by meaning)

Core principle: "Write what things *are*, not how they *look*."

✗ *PRESENTATIONAL (old way)*

* Logo *

* Article *

* Links *

✓ *SEMANTIC (HTML5 way)*

<header>Logo</header>

<main><article>Article</article></main>

<aside>Links</aside>

3.6.2. Document Outline Theory

HTML5 creates a **hierarchical outline** like a book table of contents. Browsers build this automatically:

Document Outline:

1. <h1>Main Title</h1>

1.1 <h2>Section 1</h2>

1.1.1 <h3>Subsection</h3>

1.2 <h2>Section 2</h2>

2. <h1>Next Page</h1>

Semantic containers like <section>, <article>, and <aside> are the structural building blocks that divide a document into meaningful, self-contained units. Unlike generic elements that only provide visual grouping, semantic containers communicate content purpose and establish document sections within the HTML outline algorithm. This creates a logical hierarchy that screen readers, search engines, and developers can navigate systematically.

How Semantic Containers Work in the Outline Algorithm

When a browser parses HTML5 with semantic elements, it automatically generates a **document outline** – essentially a table of contents. Semantic containers act as **sectioning roots** that reset or organize the heading hierarchy:

<section> – Thematic Grouping (Continues Parent Context)

```
<main>
  <h1>Web Technologies Course</h1>

  <section id="networking">
    <h2>Networking Fundamentals</h2>
    <!-- Belongs to course context, not independent -->

    <section id="protocols">
      <h3>TCP vs UDP</h3>
      <p>Comparison table...</p>
    </section>

    <section id="addressing">
      <h3>IP Addressing</h3>
```

```
    <p>IPv4 vs IPv6...</p>
   </section>
  </section>
</main>
```

Outline result:

1. Web Technologies Course [h1]

  1.1 Networking Fundamentals [h2]

    1.1.1 TCP vs UDP [h3]

    1.1.2 IP Addressing [h3]

<article> – Independent, Syndicable Content (Resets Headings)

Use <article> when content could be syndicated (RSS feed, republished) and makes sense standalone:

```
<main>
  <h1>Blog: Web Technology Updates</h1>

  <article id="http3">
    <header>
      <h1>HTTP/3 Now Production Ready</h1>  <!-- h1 inside article! -->
      <p>By Selver Softic • March 18, 2026</p>
    </header>
    <p>HTTP/3 with QUIC protocol...</p>
```

```
    <footer>Tags: performance, networking</footer>
  </article>

  <article id="canvas">
    <h1>Canvas API for Data Visualization</h1>  <!-- New h1 -->
    <p>Interactive charts without libraries...</p>
  </article>
</main>
```

Outline result (articles treated as first-class sections):

1. Blog: Web Technology Updates [h1]

   1.1 HTTP/3 Now Production Ready [h1 inside article]

   1.2 Canvas API for Data Visualization [h1 inside article]

Critical rule: Articles reset heading levels to <h1>, treating each as potentially independent.

<aside> – Complementary Context (Secondary Information)

<aside> holds content that **enriches but isn't essential** to the main narrative:

```
<section id="http">
  <h2>HTTP Protocol</h2>
  <p>Stateless request-response protocol...</p>

  <!-- Sidebar content - can be skipped -->
```

```
  <aside>
    <h3>Quick Reference</h3>
    <ul>
      <li>GET - Retrieve resource</li>
      <li>POST - Create resource</li>
    </ul>
  </aside>
</section>
```

Screen readers announce <aside> as "complementary" and may allow skipping.

Practical Rules for Container Selection

Decision tree:

Is content independent/reusable? → <article>

↓ No

Does it have a clear heading/theme? → <section>

↓ No

Supplementary/sidebar content? → <aside>

↓ No

Generic container →

Complete Example: Course Chapter Structure

```
<main>
  <h1>Chapter 3: HTML Structure</h1>
```

```
  <!-- Thematic course section -->
  <section>
    <h2>3.1 Semantic Elements</h2>

    <!-- Independent tutorial article -->
    <article>
      <h1>Section vs Article</h1>
      <p>Detailed comparison...</p>
    </article>

    <!-- Supplementary info -->
    <aside>
      <h3>Browser Support</h3>
      <p>All HTML5 semantic elements: 98%+ coverage</p>
    </aside>
  </section>
</main>
```

Generated outline:

1. Chapter 3: HTML Structure [h1]

   1.1 Semantic Elements [h2]

      1.1.1 Section vs Article [h1 inside article]

This **precise semantic structure** ensures perfect document outlines, accessibility flows, and SEO hierarchies – far superior to spaghetti code.

3.6.3. Element Hierarchy and Usage Rules

Semantic HTML5 elements do not exist in isolation; they form a logical hierarchy that mirrors the structure of a document, similar to chapters, sections, and paragraphs in a book. The goal is to make the HTML tree reflect the meaning of the content, so that both humans and machines can understand which parts are global, which are navigational, and which are actual content. This hierarchy is especially important for screen readers, search engines, and maintainable CSS/JavaScript.

At the top level, we distinguish between **landmark elements** that define major regions of a page and **content containers** that structure information within those regions. Inline semantic elements then enrich the text with additional meaning.

Landmark Elements (Page-Level Regions)

Landmark elements represent high-level regions that can be used by assistive technologies to offer "jump points" through the page. They are like major sections of a building: entrance, main hall, hallways, and exit.

- <*header*> – Introductory content
  - Typically contains logo, page title, and sometimes primary navigation.
  - Can appear at the **document level** and at the **section/article level**.
  - For an article, <header> often wraps the article title, author, and metadata.
- <*nav*> – Major navigation area
  - Used for key navigation blocks: main menu, footer navigation, breadcrumb.

  - Should **not** be used for every list of links; only for navigation that helps move around the site or application.
  - Assistive technologies can let users jump directly to <nav> regions.

- <*main*> – Primary content
  - Represents the core, unique content of the page.
  - There should be **exactly one <main> per document**, and it must not be a descendant of <article>, <aside>, <footer>, <header>, or <nav>.
  - Everything that is not part of the generic template (header, nav, footer, sidebar) typically belongs inside <main>.

- <*footer*> – Concluding content
  - Appears at the **document level** and optionally inside <article> or <section>.
  - Often contains copyright notices, contact information, related links, or meta information like "Last updated".
  - A footer inside an <article> refers to metadata of that article, not the whole page.

Using these landmarks consistently allows screen reader users to skip directly to “Main content”, “Navigation”, or “Footer” without traversing every element sequentially.

Now after we got the basic overview over HTML that addresses structure of the information representation we will move to the part that addresses appearance.

4. Design and Layout with CSS

4.1. Introduction to Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is the language that describes the look and layout of HTML documents, (World Wide Web Consortium [W3C], 2023). While HTML defines the structure and semantic content of a page, CSS is responsible for the entire visual presentation – from colors and fonts to the positioning of elements on the page. The decisive advantage of CSS is the consistent **separation of content (HTML) and presentation**. This greatly simplifies the maintenance of large websites, as the design of hundreds of pages can be consistently controlled by modifying a single central CSS file.

Three methods for integrating CSS

There are three ways to apply CSS styles to an HTML document, each with its specific use cases:

1. **External CSS (Preferred Method)** styles are defined in a separate .css file. This file is then included in the <head> area of each HTML page via the <link> element. This is the most efficient and maintenance-friendly method for designing entire websites.
2. *Example (index.html):*
3. *Example (mystyle.css):*
4. **Internal CSS** styles are defined directly in the <head> section of a single HTML document within a <style> element. This method is useful for page-specific style adjustments that are not needed anywhere else.

5. **Inline** CSS styles are applied directly to a single HTML element via the style attribute. This should only be used for very specific, one-time customizations, as it removes the fundamental separation of content and presentation and makes maintainability more difficult.

Cascade and priority rules

The "cascading" in the name describes that styles from different sources are applied according to a fixed hierarchy of priority. If there are several conflicting style rules for an element, the one with the highest priority wins:

1. **Inline style** (in the style attribute) has the highest priority.
2. **External and internal stylesheets** (in the <head>) have the middle priority.
3. **Browser default styles** have the lowest priority.

So an inline style always overrides a rule from an internal or external stylesheet. To determine which HTML elements these styles should be applied to, CSS uses powerful tools known as selectors.

4.2. CSS Selectors: Addressing Elements in a Targeted Manner

CSS selectors are patterns that are used to select the HTML elements to style. They are the core of CSS because they provide the precise connection between the HTML document and the style rules, (Powell, 2010).

Combinator Selectors

Combinators define the relationship between two or more simple selectors to select elements based on their position in the document tree.

Table 13. *Combinator selectors*

Combinator	Example	Description
Descendant ( )	div p	Selects all <p> elements **that are somewhere within** a element (no matter how deeply nested).
Child (>)	div > p	Selects all <p> elements that are **direct children** of a element (only one level deep).
Neighboring neighbor (+)	div + p	Selects the first <p> element **that follows immediately after** a element on the same layer.
General Neighbor (~)	div ~ p	Selects all <p> elements that follow a element on the same layer.

Pseudo-Classes

A pseudo-class defines a special state of an element, e.g. when the user hovers over it with the mouse or a form field receives focus. The syntax is selector:pseudo-class.

A classic example is the design of links in different states. **Important:** The order of these rules (L-V-H-A: Link-Visited-Hover-Active) is crucial for their correct functioning.

- a:link – a normal, unvisited link
- a:visited – a link that the user has already visited
- a:hover – a link when the user hovers over it
- a:active – a link the moment it is clicked

Other useful pseudo-classes are :focus (for focused form fields) or :first-child.

Pseudo-elements

A pseudo-element formats a specific *part* of an element, such as the first letter or a preceding decoration. The syntax is selector::p seudo-element.

- ::first-line – selects the first line of a text element.
- ::before and ::after – insert content before or after the actual content of an element that only exists in CSS.

Attribute Selectors

Attribute selectors can be used to select elements based on the presence or value of their HTML attributes.

Table 14. *Attribute selectors*

Selector	Example	Description
[Attributes]	[target]	Selects all elements that have a target attribute.
[attribute=value]	[target="_blank"]	Selects all elements whose target attribute has an exact value of "_blank".
[attribute~=value]	[title~="flower"]	Selects elements whose title attribute contains a list of words in which the word "flower" appears.
[attribute^=value]	a[href^="https"]	Selects <a> elements whose href value begins with https.
[attribute$=value]	a[href$=".pdf"]	Selects <a> elements whose href value ends with ".pdf".
[attribute*=value]	a[href*="campus02"]	Selects <a> elements whose href value contains the string "campus02".

After an element is successfully selected, it can be assigned style properties such as colors, fonts, and spacing. A basic concept for the layout is the box model.

4.3. The CSS Box Model and Units

Each HTML element on a web page is rendered by the browser as a rectangular box. The CSS box model is a fundamental concept that describes how these boxes are built and how their size and spacing affect the overall layout of the page.

Components of the Box Model

Each box consists of four layers, arranged from the inside out:

1. **Content:** The actual content of the element – the text, image, or other nested elements.
2. **Padding:** A transparent area that surrounds the content, creating a space between the content and the frame.
3. **Border:** A border that encloses the content and padding. The style, thickness and color of the frame can be defined.
4. **Margin (Outer Distance):** A transparent area outside the frame that creates a gap between this element and its adjacent elements.

Absolute and relative units of length

To define the size of the box components, CSS offers different length units that can be divided into two categories:

- **Absolute units:** These have a fixed, physical quantity and are not dependent on other elements. They are well suited for media with fixed dimensions, such as printing, but are often too rigid for screens.
  - Examples: px (pixels), cm, in (inches), pt (dots), pc (pica).
- **Relative units:** Their size is relative to another length, such as the font size of the parent element or the size of the display window (viewport). They are crucial for creating flexible and responsive designs:

- %: Relative to the corresponding property of the parent element.
- em: Relative to the font size of your *element*.
- rem (root em): Relative to the font size of the *root element* (<html>), which facilitates global scaling.
- vw, vh: 1% of the width or height of the viewport.
- vmin, vmax: 1% of the smaller or larger dimension of the viewport.

With the understanding of the box model, we can now look at specific design aspects such as background, font, and the positioning of elements in the layout.

4.4. Design of Text, Background and Layout

Designing backgrounds

The background-color property is used to set the background color of an element. Colors can be defined as color names, RGB, HEX, or HSL values to provide a wide range of design options.

Format fonts and text

- **Font-family:** This property sets the font. It is common practice to specify a list of fonts (e.g. font-family: Arial, Helvetica, sans-serif;). The browser uses the first available font from the list. At the end, there should always be a generic family (serif, sans-serif, monospace, etc.) as a fallback. "Web Safe Fonts" are fonts that are pre-installed on most operating systems.
- **Other font properties:**

Table 15. *Other font properties*

Feature	Function
font-style	Specifies the font style (for example, normal, italic).
font-weight	Determines the thickness of the font (e.g. normal, bold).

Feature	Function
font-variant	Used for small caps (small-caps).
font-size	Defines the font size (e.g. in px, em, rem).

- **Shorthand font property:** To shorten the code, several font properties can be combined into a single font declaration. The values for font-size and font-family are mandatory and must be at the end.
  - Syntax: font: [font-style] [font-variant] [font-weight] font-size[/line-height] font-family;

Dimensions and positioning

- **Dimensions:** width and height are used to festgelegt. max-width is particularly useful for responsive design, as it prevents an element from becoming wider than a certain value, while it can shrink on smaller screens.
- **Position:** The position property is fundamental to the layout and determines how an element is positioned in the document flow.

Table 16. *Dimensions and positioning*

Value	Description
Static	Default value. The element follows the normal flow of documents. The top, left, etc. properties have no effect.
Relative	The element moves relative to its normal position, but retains its original place in the layout.
absolute	The element is removed from the normal flow and positioned relative to the next positioned ancestor element.
Fixed	The element is positioned relative to the browser window and remains in its fixed position even when scrolling.
Sticky	A mixture of relative and fixed. The element behaves normally until it reaches a certain point when scrolling, where it "sticks".

Layout with float and clear

- **float**: This older layout technique takes an element out of the normal document flow and makes it "float" on the left or right edge of its container. The following content flows around the floated element.
- **clear**: Used to stop the flow of float elements. An element with clear: both; is positioned below all previous left and right float elements. This is important to fix layout issues caused by "taking out" the floated elements (known as "clearfix").

Control of overflow

The overflow property determines what happens if the content of an element is larger than the defined box of the element itself.

- visible: The content is not cut off and visibly protrudes beyond the box.
- hidden: The protruding content is cut off and is not visible.
- scroll: The content is truncated, and horizontal and vertical scrollbars are always displayed, even when they are not needed.
- auto: Scrollbars are added only when the content is actually overflowing.

Introduction to Flexbox

The Flexible Box Layout Module, or **Flexbox** for short, is a modern and powerful CSS layout model that is optimized for the design of one-dimensional layouts (i.e. rows or columns).

- A **flex container** is created by display: flex; on a parent element.
- The direct children of this container automatically become **Flex items**.
- With properties such as justify-content (alignment on the main axis), align-items (alignment on the transverse axis) and flex-direction

(direction of the main axis), the position and distribution of the items can be controlled easily and flexibly.

While HTML defines the structure and CSS defines the appearance, a third core technology is responsible for dynamic interaction on websites: JavaScript.

5. Introduction to Interactivity with JavaScript

5.1. Introduction: What is JavaScript and why is it indispensable?

JavaScript is the programming language of the web and one of the three must-have core languages for any web developer. While HTML is responsible for structuring content and CSS for its visual design, JavaScript gives life to a website by enabling interactivity and dynamic behavior. Without JavaScript, modern web applications as we know them today would be inconceivable. The content of this chapter uses insights from (Haverbeke, 2018; Resig & Bibeault, 2019, Flanagan, 2020) and web sources from W3C.

The roles of these three technologies can be clearly defined:

- **HTML → Structure:** Defines the basic building blocks of a web page, such as headings, paragraphs, images, and forms.
- **CSS → Design:** Shapes the appearance of the HTML structure, including colors, layouts, fonts, and animations.
- **JavaScript → behavior/interactivity:** Controls the dynamic behavior of the page, responds to user input, and communicates with web servers.

The versatility of JavaScript extends far beyond simple website interactions, making it one of the most widely used programming languages in modern software development. Originally designed to add interactivity to web pages, JavaScript has evolved into a powerful, full-stack language deployed across browsers, servers, and diverse application domains. Its areas of application can be grouped into the following categories:

Frontend (Client-Side)

On the client side, JavaScript is the fundamental technology for dynamic, interactive web interfaces. It executes directly in the user's browser, enabling rich user experiences without continuous server requests.

- **Dynamic websites and web applications:** JavaScript underpins single-page applications (SPAs) and progressive web apps (PWAs), where content is updated asynchronously through frameworks such as React, Angular, and Vue.js.
- **Dialog boxes and popups:** Scripting facilities allow the implementation of alert messages, confirmation dialogs, and custom modal windows for user confirmation or input.
- **Validation of form entries:** Client-side validation ensures that user input meets specified criteria (e.g., required fields, correct email format) before being transmitted to the server, improving both usability and performance.
- **Animations and dynamic menus:** Developers can implement visual transitions, dropdowns, carousels, and interactive menus using JavaScript alone or in combination with CSS, enhancing user engagement and interface feedback.
- **Display of date and time:** JavaScript can dynamically retrieve and present local or system time, timestamps, or countdowns, enabling context-aware and time-sensitive content.

Backend (Server-Side)

With the rise of **Node.js**, JavaScript can be executed on the server, supporting end-to-end application development using a single language. This architecture contributes to faster development cycles and reduced context switching.

- **Server applications:** Node.js enables JavaScript-based backends that handle business logic, API endpoints, and data processing. Frameworks such as Express.js, NestJS, and Fastify provide structured approaches to building RESTful or GraphQL services.
- **Control of web servers:** JavaScript can manage HTTP request handling, routing, session management, and integration with databases (e.g., MongoDB, PostgreSQL), as well as file-system operations and environment-specific configuration.

Other Applications

Beyond traditional web contexts, JavaScript has expanded into additional domains through cross-platform frameworks and runtime environments.

- **Mobile applications** Frameworks such as React Native and Ionic allow developers to build native-like mobile applications for both iOS and Android platforms using JavaScript, sharing a significant portion of code between platforms.
- **Games:** JavaScript-based game engines like Phaser and Babylon.js support the development of 2D and 3D browser-based games, frequently leveraging HTML5 Canvas and WebGL for rendering.
- **Presentations:** Tools such as Reveal.js enable the creation of interactive, code-driven presentations that can embed live demos, animations, or real-time data visualizations, bridging formal teaching with practical software outputs.

But how is JavaScript code actually executed? When a browser encounters JavaScript code, it passes it to its built-in JavaScript engine (e.g., V8 in Chrome). This engine processes the code in a multi-step process that ensures that the computer understands and can execute the instructions:

1. **Parsing:** The engine reads the code and breaks it down into its syntactic components (tokens).
2. **AST Creation:** From the tokens, the parser creates an "Abstract Syntax Tree" (AST), a tree-like representation of the logical structure of the code.
3. **Compilation:** A just-in-time (JIT) compiler takes the AST and converts it into highly optimized machine code.
4. **Execution:** The machine code is executed directly by the computer's processor, which enables high performance.

It's important to emphasize that **JavaScript and Java are completely different languages**, despite the similarity in names. JavaScript was developed by Brendan Eich in 1995 and adopted as an official standard in 1997 under the name **ECMAScript** (standard: ECMA-262). This standard ensures that JavaScript works consistently across browsers and environments.

Now that we understand the strategic importance of JavaScript, let's look at how it can be practically integrated into a web page and used to output data.

5.2. Putting JavaScript into Practice: Integration and Data Output

To take advantage of the dynamic capabilities of JavaScript, the code must be integrated into an HTML document. This is done using the HTML <script> tag, which signals to the browser that the content contained or linked to is to be interpreted as JavaScript code.

The function of the <script> tag is simple: it either encapsulates the JavaScript code directly or refers to an external file. In the past, the specification *type="text/javascript"* was common, but today this attribute is no longer required because JavaScript is the default language for scripts on the web.

Scripts can be placed in two primary locations within an HTML document: the <head> or <body> section. It is also possible to use any number of <script> tags in a document to logically structure the code.

External JavaScript files

For a clean code structure and better maintainability, it is common practice to outsource JavaScript code to external files with the extension .js. These files are then included in the HTML document via the src attribute of the <script> tag:

<script src="datei.js"></script>

An external script can be referenced in three ways:

- **With a full URL:** The file is loaded from another server (e.g. <script src="https://example.com/script.js"></script>).
- **With a relative file path:** The file is located in another folder on your own website (e.g. <script src="/js/script.js"></script>).
- **Without path:** The file is located in the same directory as the HTML document (e.g. <script src="script.js"></script>).

The use of external scripts offers decisive advantages:

- **Separation of HTML and JavaScript:** The HTML code remains clear and free of script logic.
- **Better readability and maintainability:** Code is stored centrally in one place and is easier to manage, especially for complex projects.
- **Caching advantages:** Browsers can cache external .js files. If a user visits another page that uses the same script file, there is no need to download it again, which reduces load times.

Example:

An external file myScript.js contains the following code:

```
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
```

This is embedded in an HTML file as follows:

```
<script src="myScript.js"></script>
```

Data Output Capabilities

JavaScript provides several methods to output data or results in the browser. The choice of method depends on the respective application.

Table 17. *Data output capabilities*

Method	Description & Use Case
innerHTML	Writes or modifies the HTML content of a specific element. Ideal for dynamically displaying data on the website, e.g. after a calculation or a user interaction.
document.write()	Writes directly to the HTML output stream. **Note:** If this method is invoked after the page has fully loaded, it will overwrite the entire existing document content. It should therefore only **be used** for testing.
window.alert()	Displays a modal dialog box (an "alert window") with a message and an OK button, useful for important notifications or simple debugging.
console.log()	Write a message in the browser's developer console. This is the preferred method for debugging during development because it does not interfere with the user.

To effectively use these methods and the language itself, we next need to understand the basic "grammar" of JavaScript: its syntax, variables, and data types.

5.3. JavaScript Grammar: Syntax, Variables, and Data Types

Like any language, JavaScript follows a fixed set of rules – its syntax. A clear and correct syntax is the basis for any functioning program. It defines how instructions are structured, values are stored, and data is processed.

5.3.1. Instructions and Comments

A **JavaScript statement** is a command that the JavaScript engine executes. It is made up of values, operators, expressions, keywords and comments. Several statements are separated from each other by a semicolon (;)). Although the semicolon is optional in many cases, its use is highly recommended to avoid ambiguity in the code.

Comments are used to explain the code and improve its readability. They are ignored by the JavaScript engine and are not executed.

- **Single-line comments** begin with //. Anything that follows after that on the same line is treated as a comment.
- **Multi-line comments** begin with /* and end with */.

```
//This is a one-line comment.

let x = 5; Comments at the end of a line are also possible.

/*

This is a

multi-line commentary, which is
```

extends over several lines.

*/

5.3.2. Values: Literals and Variables

JavaScript distinguishes between two types of values:

1. **Fixed values (literals):** These are values that are written directly into the code and do not change.
    - **Number literals:** Can be written with or without decimal places (e.g. 10.50, 1001).
    - **String literals:** Are text sequences that are enclosed in single (') or double (") quotation marks (e.g. 'John Doe', 'Jane Doe').
2. **Variable Values:** These are containers for storing data values, the contents of which can be changed during program execution.

5.3.3. Variables and Constants in Detail

Variables are the fundamental concept for storing data. In modern JavaScript, they are declared with the keywords let and const, (Ecma International, 2025). The older keyword var has a so-called *function scope*, which in the past often led to errors that were difficult to understand. The introduction of let and const with *Block Scope* solves this problem by restricting variables to the block ({...}) in which they were defined. This makes the code more secure and predictable.

- let declares a variable whose value can be reassigned.
- const declares a constant whose value cannot be changed after the first assignment.

The following table compares the properties of the three declaration keywords:

Table 18. *Variables and constants*

Comparison of Declaration Keywords	**var**	**let**	**const**
Scope	Functional scope	Block-Scope	Block-Scope
Redeclare	Allowed	Not allowed	Not allowed
Reassign	Allowed	Allowed	Not allowed
Hoisted (Raised to the Beginning)	Yes	No	No
Binds this	Yes	No	No

Instructor Tip: For modern JavaScript code, the rule is: Use const as the default. Only if you know that the value of a variable needs to be reassigned, do you use let. The var keyword should no longer be used in new code.

There are two special rules for const:

1. The value cannot be reallocated.
2. A constant must be initialized when it is declared (i.e., it must be given a value immediately).

5.3.4. Naming Rules and Conventions

Fixed rules apply to the naming of variables and constants (so-called identifiers):

- Names must begin with a letter (a-z, A-Z), a dollar sign ($), or an underscore (_).
- Subsequent characters may also contain digits (0-9).
- Names must not begin with a number.

In addition, JavaScript is **case-sensitive**, which means that upper and lower case letters make a difference. A variable named lastName is not the same as lastname.

```
let lastName = "Doe";

let lastname = "Peterson";
```

Two different variables have been declared here.

For variable names that consist of several words, the **"camel case"** notation has become established. In JavaScript, the **"Lower Camel Case"** variant is the standard for variables and functions: the first word is lowercase, and the first letter of each following word is capitalized (e.g., firstName, masterCard).

5.3.5. Data Types at a Glance

Every variable in JavaScript has a data type. A distinction is made between primitive data types and the reference type Object.

Primitive data types:

- **String:** A string (text), e.g. "Hello world".
- **Number:** A number, whole or with decimal places, e.g. 16 or 7.5.
- **BigInt:** For very large integers.
- **Boolean:** A truth value, either true or false.
- **Undefined:** A variable that has been declared but has not yet been assigned a value.
- **Null:** Represents the intentional absence of an object value.
- **Symbol:** A unique and unchanging value.

Reference Type:

- **Object:** A collection of key-value pairs. Arrays, dates, and other complex structures are also objects.

```
//Number

let length = 16;

//String

let color = "Yellow";

//Boolean

let x = true;

Object

const person = {firstName:"John", lastName:"Doe"};

Array (a special type of object)

const cars = ["Saab", "Volvo", "BMW"];
```

These basic building blocks are now combined into logical expressions and calculations using operators.

5.4. Logic and Calculations: Operators and Expressions

Operators are the tools in JavaScript to process, compare, and combine values. They enable calculations and logical decisions. A combination of values, variables, and operators that is evaluated to a single value is called an **expression**.

- 5 * 10 is an expression that evaluates to 50.
- "John" + " " + "Doe" is an expression that evaluates to "John Doe".

The most important operators can be divided into the following categories:

Arithmetic Operators

These operators perform mathematical calculations.

Table 19. *Operators for mathematical calculations.*

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulo (remainder of a division)
++	Increment (increased by 1)
--	Decrement (decreased by 1)

Assignment Operators

These operators assign values to variables.

Table 20. *Operators for assigning values*

Operator	Long form
=	x = y
+=	x = x + y
-=	x = x - y
*=	x = x * y

Operator	Long form
/=	x = x / y
%=	x = x % y
**=	x = x ** y

Comparison Operators

These operators compare two values and return a Boolean value (true or false).

Table 21. *Operators for comparing values*

Operator	Description
==	Equal (value comparison only)
===	Strictly the same (value **and** type comparison)
!=	Unequal
!==	Strictly unequal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
?	Ternary operator (short for if-else)

Important note: In professional development, the use of the strict comparison operator === is highly recommended, as it ensures type safety and eliminates a common source of error.

Logical Operators

These operators are used to combine logical expressions. They are the basis for conditional instructions.

Table 22. *Logical operators*

Operator	Description
&&	Logical AND (becomes true if both sides are true)
!	Logical NOT (inverts the truth value)

Concatenation of strings

The + operator has a special function when applied to strings: it is used to **concatecate** strings.

let text1 = "John";

let text2 = "Doe";

let text3 = text1 + " " + text2; The result is "John Doe"

Now that we know how to process values and formulate logical conditions, we can use these building blocks to control the flow of a program in a targeted manner.

5.5. Controlling Program Flow: Control Structures and Loops

Control structures are a fundamental part of programming. They make it possible to execute code only under certain conditions or repeatedly. This allows programs to react to different situations and handle complex tasks efficiently.

5.5.1. Conditional Statements

Conditional statements execute blocks of code only when a specified condition is met.

The if... else statement This is the most common form of conditional statement. The if block is executed if the condition is true. Optionally, an else block can be appended that will execute if the condition is false.

```
if (hour < 18) {
  greeting = "Good day";
} else
 { greeting = "Good evening"; }
```

The switch statement The switch statement is a useful alternative to long if... else if... else chains. It evaluates an expression and compares the result with different case values. If a case is true, the corresponding code block is executed.

```
let x = "0";
switch (x) {
  Case 0:
    text = "Off";
    break;
  Case 1:
    text = "On";
    break;
  Default:
    text = "No value found";
}
```

5.5.2. Code Retry Loops

Loops are used to execute a block of code multiple times as long as a certain condition is met.

The for loop The classic for loop is ideal when the number of repetitions is known in advance. It consists of an initialization, a condition, and an increment expression.

```
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

The for... in- and for...

These two types of loops are used to iterate over the elements of objects or arrays, but in different ways.

- **for... in:** Iterates over the **properties (keys/indexes)** of an object. For an array, this would be the index numbers 0, 1, 2, ...
- **for... of:** Iterates over the **values** of an iterable object (such as an array or a string).

Best Practice: Use for... of for arrays and other iterable data structures, as it directly returns the values. for... in is intended for iteration over the properties of generic objects and should be avoided in arrays as it can lead to unexpected behavior.

In order to organize code even better and avoid duplication, it makes sense to encapsulate recurring logic in reusable blocks, so-called functions.

5.6. Organizing and Reusing Code: Features

Functions are one of the most important tools for developing structured, maintainable, and reusable code. A function is a self-contained block of code that performs a specific task and can be called multiple times if necessary. You can think of it like a recipe: it defines a series of steps to create a finished dish (the return value) from ingredients (parameters).

There are numerous benefits to using features:

- **Code reuse:** Once defined, code can be used as often as you like.
- **Flexibility:** By passing different values (arguments), a function can deliver different results.
- **Structuring:** Complex programs are broken down into smaller, logical units, which increases clarity.
- **Maintainability:** Changes only need to be made at a central location in the function definition.

A function is defined with the keyword function, followed by a name and parentheses (). The code to be executed is enclosed in curly brackets {}.

function name(parameter1, parameter2) {

Code to execute

}

A clear distinction is made between **parameters** and **arguments** :

- **Parameters:** Are the wildcard names (the "ingredient list") that are listed in the function definition within the parentheses.
- **Arguments:** Are the actual values (the "concrete ingredients") that are passed to the function when it is called.

A function is executed when it is called. This can be done in several ways: through an event (e.g. a mouse click), directly from the code, or automatically (in the case of self-calling functions). The crucial part of the call is the () operator. Without it, the function is not executed, but only referenced as an object.

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
```

let value1 = toCelsius(77); Executes the function, value1 is 25

let value2 = toCelsius; References only the function, does not execute it

The return statement terminates the execution of a function and optionally returns a value to the caller. The code after a return statement is no longer reached.

Variables declared within a function are **local variables**. Their scope is limited to the function, i.e. they can only be used within this function. This prevents name conflicts with variables outside the function.

```
carName can NOT be accessed here
function myFunction() {
  let carName = "Volvo";
 //  carName can be accessed here
}
```

carName can NOT be accessed here

Functions help us organize code. In order to logically bundle data, JavaScript provides powerful tools in the form of objects and arrays.

5.7. Structuring Complex Data: Objects and Arrays

While primitive data types store individual values such as a number or a string, they are often not sufficient to represent more complex, related data. This is where objects and arrays come into play, making it possible to structure and group data in a meaningful way.

5.7.1. Objects: The Cornerstone of JavaScript

An object in JavaScript is a container for named values called **properties** and actions called **methods**. You can think of an object like an object from the real world.

The analogy of a **car object** illustrates this:

- **Properties** describe the condition of the car: car.name = "Fiat", car.model = 500, car.color = "white".
- **Methods** describe what the car can do: car.start(), car.drive(), car.stop().

There are several ways to create an object (Object Literal, new Object(), Object Constructor). However, using **the object literal syntax** is considered a "best practice" because it is the most readable, easiest, and fastest.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

There are two ways to access an object's properties:

- **Point notation:** person.lastName; // Returns "Doe"
- **Parenthesis notation:** person["lastName"]; // Also results in "Doe"

A **method** is a function that is stored as a property in an object. It is defined as a property and called with the () operator.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

Calling the method

```
let name = person.fullName();
```

name is now "John Doe"

5.7.2. Arrays: Ordered Lists of Data

An array is a special type of variable that can store multiple values in an ordered list. Instead of declaring a separate variable for each value, you can bundle them into a single array.

The simplest way to create an array is to use the **array literal**:

```
const cars = ["Saab", "Volvo", "BMW"];
```

The elements of an array are accessed by their **index number**. It is important that the count starts at **0** .

- cars[0] accesses the first element ("Saab").
- cars[1] accesses the second element ("Volvo").

Arrays have a variety of useful built-in methods to manipulate and iterate through data.

Basic array methods: These methods are used to manipulate an array directly – adding, removing, or extracting elements.

- length: Returns the number of elements.
- toString(): Converts the array to a comma-separated string.
- push(): Adds an element at the end.
- pop(): Removes the last element.
- shift(): Removes the first element.
- unshift(): Adds an element at the beginning.
- concat(): Joins multiple arrays.
- slice(): Copies part of the array to a new array.

Iteration Methods: These methods perform an operation on each element in the array and are a core concept of functional programming in JavaScript.

- forEach(): Executes a function for each element.
- map(): Creates a new array with the results of a function applied to each element.
- flatMap(): Combines map() with flattening the resulting array.

- filter(): Creates a new array with all the elements that pass a test.
- reduce(): Reduces the array to a single value.
- reduceRight(): Applies a function against an accumulator and any element in the array (from right to left).
- every(): Checks if all elements pass a test.
- some(): Checks if at least one element passes a test.
- from(): Creates a new array from an iterable object.
- keys(): Returns an array iterator object that contains the keys for each index in the array.
- entries(): Returns a new array iterator object that contains the key-value pairs for each index in the array.
- with(): Updates an element at a given index (ES2023).
- Spread (...): Extends an array in places where multiple elements are expected.

Now that we know how to structure data and code, we can link them to user interactions to make a web page truly dynamic.

5.8. Creating Interactivity: An Introduction to HTML Events

Events are the bridge between the user and the JavaScript code. They are actions that take place on a web page – either by the user or the browser itself – and to which we can react with JavaScript, (Flanagan, 2020) .

For example, an event can be:

- A web page is fully loaded.
- The value in an input field has been changed.

- A button was clicked.
- The mouse was moved over an element.

JavaScript allows us to "listen" for these events and execute code in response. The easiest way to do this is to use **HTML event attributes**. The syntax for this is:

<element event="some JavaScript">

Where event is the name of the event (e.g. onclick, onmouseover) and the value of the attribute is the JavaScript code that should be executed.

The following examples show how to respond to a button's onclick event:

1. Direct code execution in the attribute:

<button onclick="document.getElementById('demo').innerHTML = Date()">

The time is?

</button>

2. Call a predefined function: This is the cleaner and recommended method because it separates HTML and JavaScript logic better.

<button onclick="displayDate()">The time is?</button>

<script>

function displayDate() {

document.getElementById('demo').innerHTML = Date();

}

</script>

The basics covered here – from syntax to data structures to events – form a solid foundation for development with JavaScript and the examination of further concepts.

5.9. Beyond Basics: Modern Web Extensions

While this introductory text establishes core foundations from TCP/IP to JavaScript, contemporary web development extends these layers into advanced paradigms like serverless architectures, edge computing, modern frameworks, and high-performance runtimes. These build directly on HTTP protocols (Ch. 2), frontend standards (Ch. 3-5), and security (Ch. 1.6), addressing scalability and performance in cloud-native environments.

Serverless Architectures

Serverless computing (e.g., AWS Lambda, Vercel) eliminates server management by executing code in response to HTTP events, auto-scaling across distributed systems. Leverages HTTP/3 for low-latency triggers; stateless functions align with HTTP design (Fielding et al., 2022; Barbera et al., 2024). Ideal for APIs/microservices; persistence via external stores (e.g., DynamoDB).

Edge Computing

Processes logic at network periphery (e.g., Cloudflare Workers, Fastly) rather than central data centers, minimizing DNS/HTTP latency (Sellami et al., 2023). Runs JS/Wasm near users; enhances caching (Ch. 2.3) and TLS (Ch. 1.6) for global apps.

Modern Frameworks and WebAssembly

Frameworks like React, Vue, or Svelte build on JS DOM/events (Ch. 5) with component models and virtual DOMs for efficient UIs (W3C, 2025). WebAssembly

(Wasm) compiles high-perf languages (Rust/C++) to browser, extending HTML5 semantics (Ch. 3.6); enables PWAs with offline Service Workers.

Cross-Cutting Security

NIST controls (SP 800-53 Rev. 5) guide secure distributed web: zero-trust, encryption at rest/transit (NIST, 2023). Integrates cookie attributes (Ch. 2.2), HTTPS, and SameSite for framework/edge deployments.

These paradigms evolve your layered model for 2026 realities like AI-driven apps and IoT. For implementation, consult expanded reference

Table Index

Bibliography

Barth, A. (2011). HTTP state management mechanism (RFC 6265). Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/html/rfc6265

Ecma International. (2025). ECMAScript 2025 language specification (16th ed., ECMA-262). https://ecma-international.org/publications-and-standards/standards/ecma-262/

Fielding, R. T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616). Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/html/rfc2616

Flanagan, D. (2020). JavaScript: The definitive guide (7th ed.). O'Reilly Media.

Haverbeke, M. (2018). Eloquent JavaScript: A modern introduction to programming (3rd ed.). No Starch Press. https://eloquentjavascript.net/

Kurose, J. F., & Ross, K. W. (2020). Computer networking: A top-down approach (8th ed.). Pearson.

Mockapetris, P. (1987). Domain names – implementation and specification (RFC 1035). Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/html/rfc1035

Mozilla Developer Network (MDN). (2025). Web standards model. https://developer.mozilla.org/en-US/docs/Learn_web_development/Getting_started/Web_standards/The_web_standards_model

Powell, T. A. (2010). HTML & CSS: The complete reference (5th ed.). McGraw-Hill Education.

Resig, J., & Bibeault, B. (2019). Secrets of the JavaScript ninja (2nd ed.). Manning Publications.

World Wide Web Consortium (W3C). (2024). HTML5: A vocabulary and associated APIs for HTML and XHTML. https://www.w3.org/TR/html5/

World Wide Web Consortium (W3C). (2023). Cascading Style Sheets Level 2 Revision 2 (CSS 2.2). https://www.w3.org/TR/CSS22/

Fielding, R. T., et al. (2022). HTTP/3: QUIC-based HTTP. IETF RFC 9114. (Extends HTTP evolution beyond HTTP/2).

Barbera, E., et al. (2024). Serverless Computing Architectures: Principles and Patterns. Springer. (Intro to serverless as a web backend paradigm).

N. F. Pub. 800-53 Rev. 5 (2023). Security and Privacy Controls for Information Systems. NIST. (Modern security frameworks for web/distributed systems).

Sellami, S., et al. (2023). Edge Computing for Web Applications: A Survey. IEEE Transactions on Services Computing. (Emerging edge paradigms linked to web scalability).

W3C. (2025). WebAssembly System Interface (WASI). W3C Candidate Recommendation. (Modern framework for high-performance web apps).

Appendix: List of abbreviations

Abbreviation	Full Name / Short Description
2FA	Two-Factor Authentication
ACID	Atomicity, Consistency, Isolation, Durability (properties of secure DB transactions)
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CD	Continuous Delivery / Deployment (Automated Rollout)
CDN	Content Delivery Network
CI	Continuous integration (automatic merging and testing)
CORS	Cross-Origin Resource Sharing (Control Access Between Domains)
CRUD	Create, Read, Update, Delete (basic operations on data)
CSP	Content Security Policy (protection against unwanted scripts)
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets (Layout and Design)
DBMS	Database Management System
DNS	Domain Name System (Translates domain names into IP addresses)
DNSSEC	DNS Security Extensions
DOM	Document Object Model (Structured representation of HTML in the browser)
HTML	HyperText Markup Language (structure of the website)
HTTP	Hypertext Transfer Protocol (standard protocol for web data)
HTTPS	Hypertext Transfer Protocol Secure (HTTP with TLS encryption)
IMAP	Internet Message Access Protocol (reading mails on the server)

IP	Internet Protocol (addressing and routing on the Internet)
IRI	Internationalized Resource Identifier (extension of the URI, supports Unicode)
ISP	Internet Service Provider
JS	JavaScript (Interactive Web Logic)
JSON	JavaScript Object Notation (data format for exchange)
JSON-RPC	JSON Remote Procedure Call (API via RPC with JSON)
JWT	JSON Web Token (Token for Authentication)
MIME	Multipurpose Internet Mail Extensions (attachments and formatting in emails)
MITM	Man in the Middle (attack by intercepting data)
MVC	Model View Controller (Architecture for Web Apps)
NoSQL	Not Only SQL (Non-relational Databases)
ORM	Object-relational mapping (connecting objects to databases)
POP3	Post Office Protocol v3 (Downloading Emails)
REST	Representational State Transfer (API Architecture Style)
SEO	Search Engine Optimization
SMTP	Simple Mail Transfer Protocol (Versand von E-Mails)
SOAP	Simple Object Access Protocol (Strict API Protocol)
SPA	Single Page Application (Dynamische Einseiten-Webapp)
SQL	Structured Query Language (language for relational DBs)
SSL	Secure Sockets Layer (predecessor of TLS)
SSO	Single Sign-On (One Login for Multiple Services)
SVG	Scalable Vector Graphics (vector graphics for the web)
TCP	Transmission Control Protocol (Reliable Data Transfer)
TLS	Transport Layer Security (encryption for network connections)
UDP	User Datagram Protocol
URI	Uniform Resource Identifier

URL	Uniform Resource Locator (address of a resource on the web)
URN	Uniform Resource Name
VM	Virtual Machine
WAF	Web Application Firewall (protection layer for web apps)
WWW	World Wide Web
XSS	Cross-Site Scripting

Closing Remarks

The covered fundamentals – from network protocols to interactive JavaScript applications – form the backbone of professional web development. Practice these concepts in your own projects, experiment with frameworks like React or Vue.js, and always prioritize security, accessibility, and performance.

The web technology landscape evolves quickly: HTTP/3, PWAs, and AI-driven development await. Use this foundation to create innovative solutions.

Best wishes for your projects!

FH-Prof. DI Dr. techn. Selver Softić, BSc
CAMPUS 02 University of Applied Sciences, Graz

About the Author

Dr. Selver Softic, is a distinguished research scientist, lecturer, and professor at CAMPUS 02 University of Applied Sciences in Graz, Austria. He authored the book "Introduction to the Basics of Web Technologies," drawing from his extensive teaching experience in web development, cloud computing, and digital transformation.

Professional Background

Selver also coordinates R&D projects at CAMPUS 02, leading EU-funded initiatives in smart production, data science, IoT, big data, and business informatics. His career includes roles as Senior Researcher at Virtual Vehicle Competence Center, Technology Consultant at Infonova, and Research Associate at JOANNEUM RESEARCH, focusing on semantic web, knowledge management, and intelligent systems.

Education and Expertise

He holds a PhD from Graz University of Technology's Doctoral School of Informatics, an MSc and BSc in Information and Computer Engineering from the same institution, and was a visiting researcher at Universiteit Gent's Data Science Lab. Expertise spans AI, visual analytics, semantic web, JavaScript, J2EE, Python, and web standards like XML, XPATH, XSLT, HTML5, CSS, and HTTP.

Contributions and Interests

Selver has led several EU projects and contributed to semantic methodologies in automotive lifecycles. Beyond academia, he pursues topics such as agentic AI, learning analytics, and human-centric technology.

ISBN 978-606-37-2971-3